

WRDC-TR-90-8007
Volume V
Part 1

AD-A250 448



INTEGRATED INFORMATION SUPPORT SYSTEM (IISS)
Volume V - Common Data Model Subsystem
Part 1 - CDM Administrator's Manual

M. Apicella, R. Palumbo, S. Singh

Control Data Corporation
Integration Technology Services
2970 Presidential Drive
Fairborn, OH 45324-6209

DTIC
ELECTE
MAY 08 1992
S B D

September 1990

Final Report for Period 1 April 1987 - 31 December 1990

Approved for Public Release; Distribution is Unlimited

MANUFACTURING TECHNOLOGY DIRECTORATE
WRIGHT RESEARCH AND DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6533

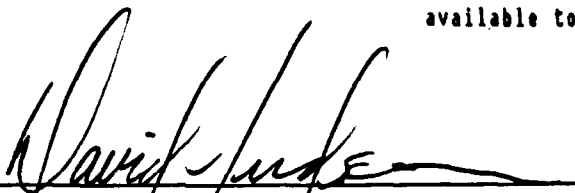


NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever, regardless whether or not the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data. It should not, therefore, be construed or implied by any person, persons, or organization that the Government is licensing or conveying any rights or permission to manufacture, use, or market any patented invention that may in any way be related thereto.


This technical report has been reviewed and is approved for publication.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations


DAVID L. JUDSON, Project Manager
WRDC/MTI
Wright-Patterson AFB, OH 45433-6533

25 July 91
DATE

FOR THE COMMANDER:


BRUCE A. RASMUSSEN, Chief
WRDC/MTI
Wright-Patterson AFB, OH 45433-6533

25 July 91
DATE

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WRDC/MTI, Wright-Patterson Air Force Base, OH 45433-6533 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

| REPORT DOCUMENTATION PAGE | | | | |
|--|--------------------------------------|--|-----------------------|---|
| 1a. REPORT SECURITY CLASSIFICATION Unclassified | | 1b. RESTRICTIVE MARKINGS | | |
| 2a. SECURITY CLASSIFICATION AUTHORITY | | 3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution is Unlimited. | | |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | | | | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) UM 620341001 | | 5. MONITORING ORGANIZATION REPORT NUMBER(S) WRDC-TR- 90-8007 Vol. V, Part 1 | | |
| 6a. NAME OF PERFORMING ORGANIZATION Control Data Corporation; Integration Technology Services | | 6b. OFFICE SYMBOL (if applicable) WRDC/MTI | | 7a. NAME OF MONITORING ORGANIZATION WRDC/MTI |
| 6c. ADDRESS (City, State, and ZIP Code) 2970 Presidential Drive Fairborn, OH 45324-6209 | | 7b. ADDRESS (City, State, and ZIP Code) WPAFB, OH 45433-6533 | | |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION Wright Research and Development Center, Air Force Systems Command, USAF | | 8b. OFFICE SYMBOL (if applicable) WRDC/MTI | | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUM. F33600-87-C-0464 |
| 8c. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB, Ohio 45433-6533 | | 10. SOURCE OF FUNDING NOS. | | |
| 11. TITLE (Include Security Classification) See Block 19 | | PROGRAM ELEMENT NO. 78011F | PROJECT NO. 595600 | TASK NO. F95600 WORK UNIT NO. 20950607 |
| 12. PERSONAL AUTHOR(S) Control Data Corporation: Apicella, M. L., Palumbo, R., and Singh, S. | | | | |
| 13a. TYPE OF REPORT Final Report | 13b. TIME COVERED 4/1/87-12/31/90 | 14. DATE OF REPORT (Yr., Mo., Day) 1990 September 30 | | 15. PAGE COUNT 222 |
| 16. SUPPLEMENTARY NOTATION WRDC/MTI Project Priority 6203 | | | | |
| 17. COSATI CODES | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify block no.) | | |
| FIELD | GROUP | SUB GR. | | |
| 1308 | 0905 | | | |
| 19. ABSTRACT (Continue on reverse if necessary and identify block number) | | | | |
| <p>This document is the Common Data Model Administrator's User Manual. Its purposes are several and include:</p> <ul style="list-style-type: none"> o Describing the philosophical and practical objectives of the CDM Administrator. o Discussing the CDM, its design, and its role in the IISS environment. o Describing the steps necessary to entering and maintaining data kept in the CDM. <p>Block 11 - INTEGRATED INFORMATION SUPPORT SYSTEM (IISS) Vol V - Common Data Model Subsystem Part 1 - CDM Administrator's Manual</p> | | | | |
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED x SAME AS RPT. DTIC USERS | | 21. ABSTRACT SECURITY CLASSIFICATION Unclassified | | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL David L. Judson | | 22b. TELEPHONE NO. (Include Area Code) (513) 255-7371 | | 22c. OFFICE SYMBOL WRDC/MTI |

FOREWORD

This technical report covers work performed under Air Force Contract F33600-87-C-0464, DAPro Project. This contract is sponsored by the Manufacturing Technology Directorate, Air Force Systems Command, Wright-Patterson Air Force Base, Ohio. It was administered under the technical direction of Mr. Bruce A. Rasmussen, Branch Chief, Integration Technology Division, Manufacturing Technology Directorate, through Mr. David L. Judson, Project Manager. The Prime Contractor was Integration Technology Services, Software Programs Division, of the Control Data Corporation, Dayton, Ohio, under the direction of Mr. W. A. Osborne. The DAPro Project Manager for Control Data Corporation was Mr. Jimmy P. Maxwell.

The DAPro project was created to continue the development, test, and demonstration of the Integrated Information Support System (IISS). The IISS technology work comprises enhancements to IISS software and the establishment and operation of IISS test bed hardware and communications for developers and users.

The following list names the Control Data Corporation subcontractors and their contributing activities:

SUBCONTRACTOR

ROLE

| | |
|--------------------------|--|
| Control Data Corporation | Responsible for the overall Common Data Model design development and implementation, IISS integration and test, and technology transfer of IISS. |
| D. Appleton Company | Responsible for providing software information services for the Common Data Model and IDEF1X integration methodology. |
| ONTEK | Responsible for defining and testing a representative integrated system base in Artificial Intelligence techniques to establish fitness for use. |
| Simpact Corporation | Responsible for Communication development. |

| | |
|--------------------|--|
| Accession For | |
| NTIS GRA&I | <input checked="checked" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution/ | |
| Availability Codes | |
| Dist | Avail and/or Special |
| A-1 | |

Structural Dynamics
Research Corporation

Responsible for User Interfaces,
Virtual Terminal Interface, and Network
Transaction Manager design,
development, implementation, and
support.

Arizona State University

Responsible for test bed operations
and support.

Table of Contents

| | | <u>Page</u> |
|------------|--|-------------|
| SECTION 1. | INTRODUCTION | 1-1 |
| 1.1 | Managing Data as a Corporate Resource | 1-1 |
| SECTION 2. | CDM OVERVIEW | 2-1 |
| 2.1 | The Fundamental Approach | 2-1 |
| 2.1.1 | The Three Schema-Architecture | 2-1 |
| 2.1.2 | Representation of the Three Types of Schemas | 2-7 |
| 2.1.3 | Integration Methodology | 2-7 |
| 2.1.4 | Contributions to IRRIASSPA | 2-10 |
| 2.2 | Basic Components of the Design..... | 2-10 |
| 2.2.1 | The CDM Database | 2-11 |
| 2.2.2 | CDM1..... | 2-11 |
| 2.2.3 | The CDM Processor..... | 2-12 |
| SECTION 3. | RESPONSIBILITIES OF THE CDM ADMINISTRATOR | 3-1 |
| 3.1 | Establishing Data Standards | 3-1 |
| 3.2 | Maintaining the CDM | 3-1 |
| 3.3 | Protecting the CDM | 3-1 |
| 3.4 | Facilitating Use of the CDM | 3-1 |
| SECTION 4. | MAINTAINING THE CONCEPTUAL SCHEMA ... | 4-1 |
| 4.1 | Methodology Overview | 4-1 |
| 4.1.1 | CS Structure | 4-1 |
| 4.1.2 | Basic Approach | 4-3 |
| 4.1.3 | Modeling Forms | 4-4 |
| 4.2 | Building the Initial CS | 4-15 |
| 4.2.1 | Phase 0: Starting the Project | 4-15 |
| 4.2.2 | Phase 1: Defining Entity Classes | 4-18 |
| 4.2.3 | Phase 2: Defining Relation Classes..... | 4-20 |
| 4.2.4 | Phase 3: Defining Key Classes | 4-22 |
| 4.2.5 | Phase 4: Defining Nonkey Attribute Classes | 4-29 |
| 4.3 | Expanding the CS | 4-30 |
| 4.3.1 | Phase 0: Starting the Project | 4-31 |
| 4.3.2 | Phase 1: Defining Entity Classes | 4-33 |
| 4.3.3 | Phase 2: Defining Relation Classes..... | 4-34 |
| 4.3.4 | Phase 3: Defining Key Classes | 4-36 |
| 4.3.5 | Phase 4: Defining Nonkey Attribute Classes | 4-46 |

Table of Contents

| | | <u>Page</u> |
|------------|--|-------------|
| SECTION 5. | MAINTAINING THE CDM | 5-1 |
| 5.1 | Methodology Overview | 5-1 |
| 5.1.1 | Generic NDDL Commands..... | 5-1 |
| 5.1.2 | Transaction NDDL Commands..... | 5-2 |
| 5.2 | Loading the Initial CS Description | 5-3 |
| 5.2.1 | Loading Domains..... | 5-7 |
| 5.2.2 | Defining the Model..... | 5-7 |
| 5.2.3 | Loading Attribute Classes..... | 5-8 |
| 5.2.4 | Loading Entity Classes..... | 5-10 |
| 5.2.5 | Loading Key Classes and Relation Classes..... | 5-12 |
| 5.3 | Modifying/Deleting CS Objects..... | 5-15 |
| 5.3.1 | Domain Class Changes | 5-15 |
| 5.3.2 | Model Changes/Deletes..... | 5-17 |
| 5.3.3 | Attribute Class Changes/Deletes..... | 5-18 |
| 5.3.4 | Entity Class Changes/Deletes..... | 5-19 |
| 5.3.5 | Relation Class Changes/Deletes..... | 5-21 |
| 5.4 | Modeling & Validating Tools..... | 5-23 |
| 5.5 | Reviewing the Contents of the CDM.... | 5-23 |
| SECTION 6. | MAINTAINING INTERNAL SCHEMAS AND MAPPINGS | 6-1 |
| 6.1 | Methodology Overview | 6-1 |
| 6.1.1 | Internal Schema and CS-IS Mapping Structure..... | 6-2 |
| 6.1.2 | CS-IS Mapping Modeling Forms..... | 6-16 |
| 6.2 | Loading The Initial Internal Schema | 6-40 |
| 6.2.1 | Loading The Distributed Database Environment..... | 6-40 |
| 6.2.2 | Loading User-Defined data types..... | 6-41 |
| 6.2.3 | Loading Databases..... | 6-41 |
| 6.2.4 | Loading Record Types And Data Fields..... | 6-43 |
| 6.3 | Loading the Initial CS-IS Mapping Definition..... | 6-50 |
| 6.3.1 | Loading CS to IS Mappings..... | 6-50 |
| 6.3.2 | Loading Record Unions..... | 6-51 |
| 6.3.3 | Loading Horizontal Partitions..... | 6-52 |
| 6.3.4 | Loading Transformational Algorithms | 6-52 |
| 6.4 | Modifying/Deleting IS Objects..... | 6-65 |
| 6.4.1 | Distributed Database Environment Changes..... | 6-65 |

Table of Contents

| | | <u>Page</u> |
|------------|---|-------------|
| 6.4.2 | Modifying User-Defined data types | 6-67 |
| 6.4.3 | Database Changes/Deletes..... | 6-67 |
| 6.4.4 | Record Type Changes/Deletes..... | 6-69 |
| 6.4.5 | Datafield Changes/Deletes..... | 6-70 |
| 6.4.6 | Modifying/Deleting CS-IS Mappings | 6-71 |
| 6.4.7 | Record Union Changes/Deletes..... | 6-73 |
| 6.4.8 | Horizontal Partition Changes/Deletes..... | 6-74 |
| 6.5 | Specific Considerations..... | 6-74 |
| 6.5.1 | IMS Specific Considerations..... | 6-74 |
| 6.5.2 | VSAM Specific Considerations..... | 6-83 |
| 6.5.3 | Sequential Files Specific Considerations..... | 6-83 |
| SECTION 7. | MAINTAINING EXTERNAL SCHEMAS MAPPINGS | 7-1 |
| 7.1 | Methodology Overview | 7-1 |
| 7.1.1 | External Schemas and CS-ES Mapping Structure..... | 7-1 |
| 7.1.2 | Modeling Forms..... | 7-10 |
| 7.2 | Loading the Initial ES & CS-ES Mapping Definition..... | 7-13 |
| 7.2.1 | Loading User-Defined data types..... | 7-13 |
| 7.2.2 | Loading User Views and Data Items | 7-14 |
| 7.2.3 | Loading Transformation Algorithms | 7-15 |
| 7.3 | Modifying/Deleting ES Elements and CS-ES Mappings | 7-18 |
| 7.3.1 | Modifying User-Defined data types | 7-18 |
| 7.3.2 | User View Changes/Deletes..... | 7-18 |
| APPENDIX A | GLOSSARY | A-1 |
| APPENDIX B | USEFUL REFERENCES | B-1 |

List of Illustrations

| <u>Figure</u> | <u>Title</u> | <u>Page</u> |
|---------------|---|-------------|
| 1-1 | Data as an Integral Part of the Decision-Making Process | 1-3 |
| 2-1 | Two Fundamentally Different Views of Data: Logical and Physical | 2-3 |
| 2-2 | Direct Mapping of Logical and Physical Views | 2-4 |
| 2-3 | The Three-Schema Architecture | 2-6 |
| 4-1 | Relation Classes Form | 4-6 |
| 4-2 | Relation Classes Form Example | 4-7 |
| 4-3 | Owned Attribute Classes Form | 4-11 |
| 4-4 | Owned Attribute Classes Form Example | 4-12 |
| 4-5 | Inherited Attribute Classes Form | 4-13 |
| 4-6 | Inherited Attribute Classes Form Example | 4-14 |
| 4-7 | Refinements of Nonspecific Relation Classes Example | 4-48 |
| 4-8 | Triads and Other Dual-Path Structures | 4-49 |
| 4-9 | Migration Through Two Relation Classes | 4-50 |
| 4-10 | Guidelines for Determining Key Classes of Dependent Entity Classes | 4-51 |
| 5-1 | CDM Objects..... | 5-4 |
| 5-2 | CDM Object Description..... | 5-4 |
| 5-3 | CDM Conceptual Schema..... | 5-6 |
| 5-4 | Owned Attribute Classes Form Example..... | 5-10 |
| 5-5 | Figure Entity Class Glossary Form Example | 5-12 |
| 5-6 | Inherited Attribute Classes Form Example..... | 5-14 |
| 5-7 | Relation Classes Form Example..... | 5-15 |
| 6-1 | Entity Class/Record Type Mapping | 6-3 |
| 6-2 | Join Examples | 6-8 |
| 6-3 | Join Structures | 6-11 |
| 6-4 | Record Type/Entity Class Mapping Form | 6-19 |
| 6-5 | Record Type/Entity Class Mapping Form Example | 6-20 |
| 6-6 | Record Type Join Structures Diagram | 6-21 |
| 6-7 | Record Type Join Structures Diagram Example | 6-22 |
| 6-8 | Data Field/Attribute Use Class Mapping | 6-24 |
| 6-9 | Data Field/Attribute Use Class Mapping Example | 6-25 |
| 6-10 | Set Type/Relation Class Mapping | 6-27 |

List of Illustrations

| <u>Figure</u> | <u>Title</u> | <u>Page</u> |
|---------------|---|-------------|
| 6-11 | Set Type/Relation Class Mapping Example | 6-28 |
| 6-12 | Data Field/Attribute Use Class Mapping Example | 6-31 |
| 6-13 | Record Type Join Structure Diagram Example | 6-38 |
| 6-14 | Incomplete Join Structure Example..... | 6-39 |
| 6-15 | CDM Tables Distributed Data Bases..... | 6-46 |
| 6-16 | CDM Tables Domains and Data Types for Internal Schema..... | 6-47 |
| 6-17 | CDM Tables Relational Database Internal Schema..... | 6-48 |
| 6-18 | CODASYL Internal Schema..... | 6-49 |
| 6-19 | CS to IS Entity Mapping..... | 6-54 |
| 6-20 | Record Type/Entity Class Mapping..... | 6-55 |
| 6-21 | S to IS Attribute and Relation Mapping | 6-56 |
| 6-22 | Datafield to Attribute Use Class Mapping..... | 6-58 |
| 6-23 | Set Type to Relation Class Mapping..... | 6-59 |
| 6-24 | Record Union..... | 6-60 |
| 6-25 | horizontal Partition..... | 6-61 |
| 6-26 | Complex Mapping Algorithm..... | 6-62 |
| 6-27 | IMS Internal Schema..... | 6-63 |
| 6-28 | IMS Internal Schema..... | 6-80 |
| 7-1 | Data Item/Attribute Use Class Mappings | 7-2 |
| 7-2 | Vertical Partition..... | 7-3 |
| 7-3 | Entity Joins..... | 7-4 |
| 7-4 | ES-CS Join Examples..... | 7-5 |
| 7-5 | ES-CS Join Structures..... | 7-8 |
| 7-6 | Single Entity Views..... | 7-10 |
| 7-7 | Domains and Data Types External Schema... | 7-16 |
| 7-8 | External Schema and CS/ES Mapping..... | 7-17 |

SECTION 1

INTRODUCTION

The purposes of this document are several and include:

- a) Describing the philosophical and practical objectives of the Common Data Model (CDM) Administrator;
- b) Discussing the CDM itself, its underlying design, and its role in the IISS environment;
- c) Describing in detail the steps necessary in entering and maintaining data kept in the CDM.

After reading and understanding this document, the CDM Administrator should not be able only to collect, enter, and maintain CDM-related data, but also be able to understand the reasons why such activities are performed.

The NDDL statements used to perform the actual CDM maintenance activities are described in detail in the NDDL User Guide.

1.1 Managing Data as a Corporate Resource

Managing data as a corporate resource is a philosophy about the importance of data to an organization. The approach recognizes that data are assets to be managed along with the other more generally recognized resources of an enterprise, including its personnel, inventories, capital, and so forth. Organizations spend tremendous sums of money collecting and manipulating data, trying to extract information needed to support decision making. The CDM Administrator has as one of his or her primary objectives the preservation of that continuing, substantial investment in data resources. The CDM Administrator plays a major role in protecting and properly managing that investment by managing common data rather than just managing applications that access data.

Data management includes all the activities that ensure that quality data are available to produce needed information and knowledge. The objective of data management is to keep data assets resilient, flexible, and adaptable to supporting decision-making activities in the business. Data management responsibilities include: 1) the representation, storage, and organization of data so that they can be selectively and efficiently accessed, 2) the manipulation and presentation of data so that they support the user environment effectively, and 3) the protection of data so that they retain their value.

The philosophy of the CDM recognizes that data are absolutely necessary to the decision-making cycles of organizations (Figure 1-1). Individuals must not only be able to collect and retain data for their own use, but also be able to share data and pool their knowledge resources. The ability to correlate information across traditional applications boundaries and to provide

information that supports all levels of decision making, from operational through tactical through strategic, is increasingly important as management at all levels is becoming more aware of the potential power of information systems.

The CDM provides the capability to pull the enterprise's database resources together to form an integrated, common source of information to support decision making.

The objectives of data management include the following:

- o Independence of data access from data descriptions
- o Increased data accessibility
- o Improved data integrity
- o Improved data shareability
- o Improved data resiliency
- o Improved data administration and control
- o Improved data security
- o Improved performance

The CDM Administrator needs to understand each of these objectives.

Independence between data access and data descriptions improves control over the data descriptions, facilitates standardization of data-naming conventions, and reduces the programming effort required to accommodate modified data descriptions. Data independence is perhaps the single most important factor in determining the long-range success of a data-driven environment.

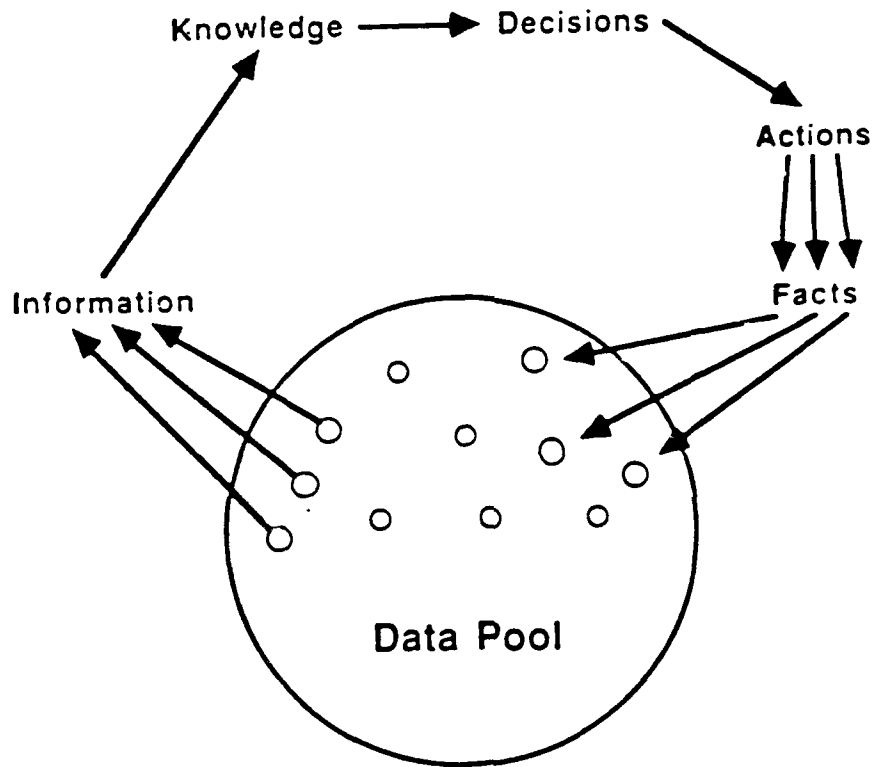


Figure 1-1. Data as an Integral Part of the Decision Making Process.

Data accessibility refers to the capability for a user to extract needed information from the data resource. Data accessibility is enhanced by user-friendly interface languages and well designed screens. Good accessibility is characterized by being able to relate data in many different ways to produce information, and by being able to represent that information in a variety of suitable forms. Data accessibility is improved by the CDM in its support of multiple access paths and retrieval sequences through the physical databases. Programming effort for data manipulation is decreased and cost-effective, general-purpose query facilities such as the NDML become possible.

Data integrity is essential to maintain the quality of the data resource. Data integrity is measured by the completeness and consistency of the data resource. Does it contain the data that are relevant to the decision-making needs of the user? Does it contain all required interrelationships among types of data, and are all consistency constraints satisfied?

Data shareability is needed to keep common data truly common. Without shareability, data proliferate and their quality becomes uncontrollable. Without shareability, data are private and personal; their quality is each individual user's responsibility. The main difficulty with this distribution and redundancy of control is that it results in no control at all. Improved shareability can be achieved by supporting multiple access paths through the physical databases, thereby enabling them to serve many diverse needs. Shareability is also achieved by separating individual user's views of the data resource from the actual physical implementation of databases.

Data shareability refers not just to database contents, but also to logic that accesses and manages data. Reduced data duplication streamlines data access, reduces the programming effort required for updating data, and reduces the potential for inconsistent data. Reduced redundancy in the data management effort improves the productivity of data processing personnel.

Data recoverability is needed to keep the data resource resilient in the wake of errors. Error conditions need to be detected and corrected. Better yet, errors should be prevented from occurring in the first place. Part of the difficulty in providing a resilient data resource is continuing to make the data available to users while recovering from errors.

The CDM Administrator should help to ensure that the data resource continues to satisfy users' information needs, even as those needs change through time. Many organizations have successfully established data administration functions to help develop and protect data assets. The CDM Administrator plays a similar role for the integrated, overall data resource.

Data security is essential to prevent unauthorized access to data. Certainly not all environments require the same, elaborate security schemes, but nearly all organizations' data assets need to have some degree of access protection. Some data are wide open to public retrieve-only access; others require

strict authentication to provide retrieval. Many databases have more stringent restrictions on accesses that will change database contents than on accesses that only read database contents.

Performance of the data resource has two facets: efficiency and effectiveness. Efficiency is a measure of how well the data system utilizes physical computer support, while effectiveness is a measure of how well the data system meets users' information needs. The characteristics are closely related; for example, a user may be totally dissatisfied with the system if response time is measured in hours rather than seconds. Response time is generally considered to be an efficiency measure, but it certainly has an impact on effectiveness.

SECTION 2

CDM OVERVIEW

2.1 The Fundamental Approach

2.1.1 The Three-Schema Architecture

A key to implementing effective data-oriented environments lies in a framework that is called the Three-Schema Architecture. This approach was proposed in the mid-1970s, then developed, and finally published in 1977 in a report from a committee of the American National Standards Institute - "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems."

The basic concepts proposed in the report have the power to lead us to more effective information resource management. They are implemented in the CDM.

The Three-Schema Architecture is based upon several fundamental facts:

- o Computers and users need to be able to view the same data in different ways
- o Different users need to be able to view the same data in different ways
- o It is (more or less) frequently desirable for users and computers to change the ways they view data
- o It is undesirable for the computer to dictate or constrain the ways that users view data

Thus, it is necessary to be able to support different types of views of a data resource. Users need to be able to work with logical representations of data, which are independent of any physical considerations of how the data are actually stored and managed on computer facilities. Users view data in terms of high-level entities, e.g., staff members, tools, vehicles, products, orders, and customers. Meanwhile, computer facilities, access methods, operating systems, and DBMSs, for example, need to be able to work with more physical representations. They view data in terms of records and files, with index structures, B-trees, linked lists, pointers, addresses, pages, and so forth.

These requirements lead us to conclude first that there are two fundamentally different types of data views: logical and physical. The logical views are user-oriented, while the physical views are computer-oriented (Figure 2-1).

A second conclusion is that there must be a mapping or transformation between the logical and physical views. After all, the ultimate objective is to enable users to gain access to their data that reside on computerized media. This mapping

might be simple if there were only one user view and one database, but that is not the real-world situation. Rather, there are multitudes of user views and commonly many (sometimes hundreds or thousands) databases in an enterprise.

Each user view could be mapped directly to the underlying databases (Figure 2-2). This solution suffers, however, when change is introduced in either type of view. If a physical database is restructured on a disk to provide more efficient performance, then the mapping to each of the user views that references that database can be affected. If a logical view is revised to present information in a somewhat different way, then the mapping to each of the referenced databases may be affected. Independence of logical and physical considerations would not have been achieved, and we would find that physical computer factors would constrain the ways that users logically view their data. This is undesirable.

Using three-schema architecture terminology, "external schemas" represent user views of data, while "internal schemas" represent physical implementations of databases. Schemas are metadata, i.e., they are data about data. As a simple example, CUSTOMER-NAME and CHARACTER (17) are metadata describing the data value CHRISTOPHER ROBIN.

To enable multiple users to share a data resource that is implemented on potentially many physical databases, we insert between the users' views and the physical views a neutral, integrated view of the data resource. This view is called a "conceptual schema" in three-schema architecture terminology. Others sometimes call it an "enterprise view."

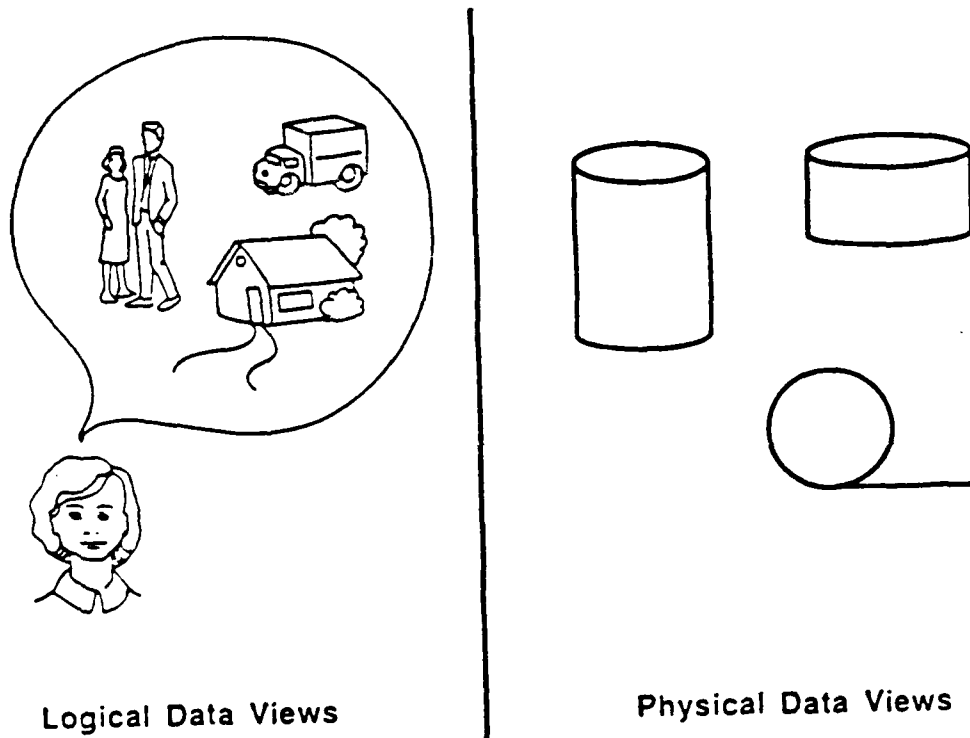


Figure 2-1. Two Fundamentally Different Views of Data: Logical and Physical

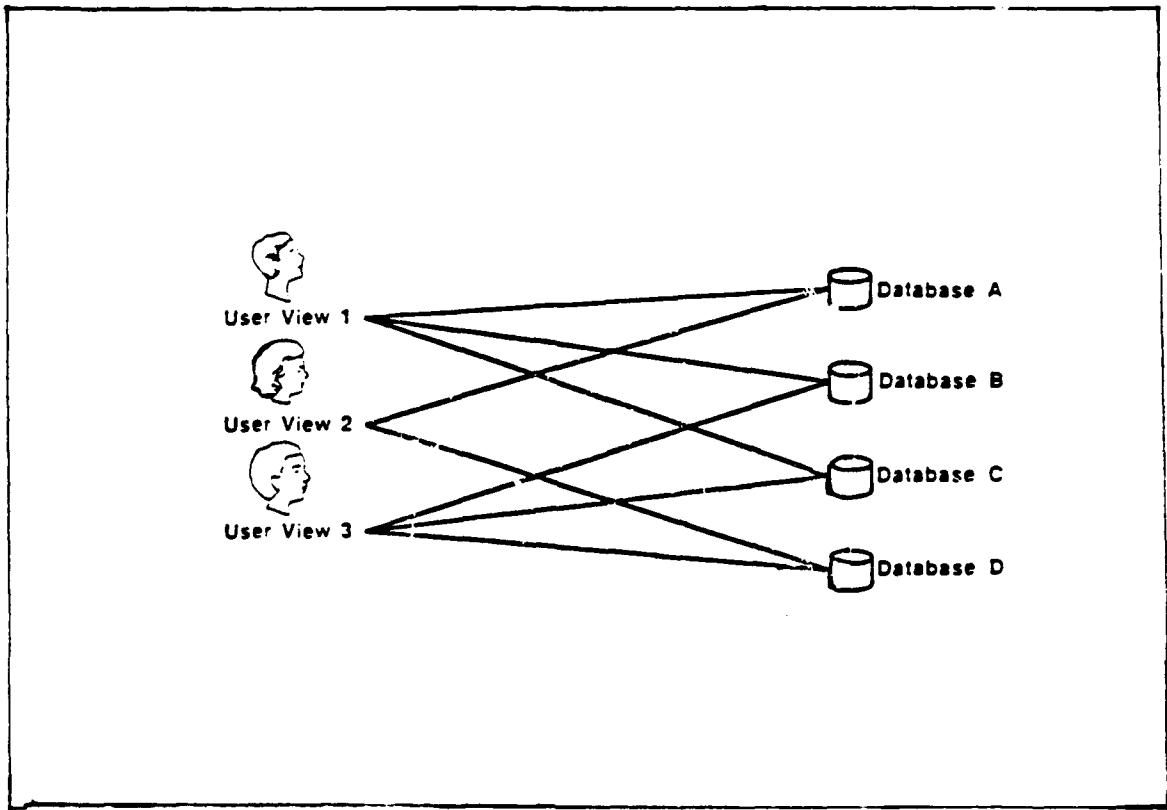


Figure 2-2. Direct Mapping of Logical and Physical Views

As the vehicle for data integration and sharing, the conceptual schema also carries metadata for enforcement of data integrity rules. It is extensible, consistent, accessible, shareable, and enables the data resource to evolve as needs change and mature.

Figure 2-3 illustrates the relationships between the three types of schemas. The schemas and the mappings between them are the mechanism for achieving both data independence and support of multiple views. An internal schema can be changed to improve efficiency and take advantage of new technical developments without altering the conceptual schema.

The conceptual schema represents knowledge of shareable data. There may be access controls and security restrictions placed upon these common data, but they are not restricted to access by only one user. The conceptual schema does not describe personal data.

The scope of the conceptual schema expands through time. The conceptual schema extension methodology continually expands the conceptual schema to include knowledge of more shared data. The external-conceptual mappings protect the external schemas and the transactions/programs that depend on them from most modifications incurred in evolving the conceptual schema.

Adding data to the integrated, common resource does not start over in defining the data resource, nor does it create another stand-alone database. Rather, development of its database must examine questions of how those data relate to what is already known by the conceptual schema. The result will be an integrated data resource whose scope is expanded gradually. It is absolute folly to approach integration of the data resources of an organization all at once; the job must be taken on piecemeal. The conceptual schema is the integrator.

The CDM contains all three types of schemas, as well as the interschema mappings. It not only documents these metadata, but also supplies appropriate metadata to support transaction processing.

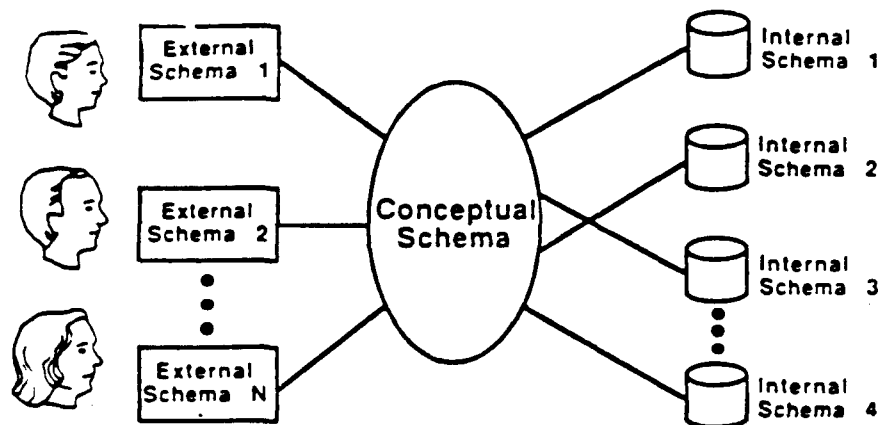


Figure 2-3. The Three-Schema Architecture: One Conceptual Schema That Provides for Integration and Independence of Many External Schemas and Many Internal Schemas

2.1.2 Representation of the Three Types of Schemas

In the IISS, the Three-Schema Architecture is implemented through the CDM facilities to store each of the three types of schemas and the interschema mappings. An appropriate representation mode has been selected for each of the three types of schemas.

The conceptual schema is represented by an IDEF1 model. The CDM stores this model in terms of entity classes, attribute classes, and relation classes.

The external schemas are represented by tables. The user views the common data resource in terms of flat, simple tables. The mappings between these tables and the IDEF1 model of the conceptual schema are part of the CDM database.

The internal schemas are represented in terms of physical database components, including record types and inter-record relationships. The CDM Processor routines convert the users' data access requests, which are phrased in terms of tables, into requests against the conceptual schema IDEF1 model, then into requests against the physical database structures described in the internal schema part of the CDM.

2.1.3 Integration Methodology

The Integration Methodology is the set of procedures and guidelines that are used to expand the conceptual schema and to increase the sphere of common data available to support users and applications. The schemas and schema mappings in the CDM are built, maintained, and accessed using the Integration Methodology and the CDM Processors. (CDMP)

The Integration Methodology is intended to guide the CDM Administrator in building and maintaining the conceptual schema and in keeping its mappings to the internal and external schemas highly accurate. This methodology consists of a set of techniques for building the conceptual schema in gradual increments, for building external and internal schemas from portions of the conceptual schema, for developing schema mappings, and for keeping these various CDM components current.

The first step in populating the CDM is to select a portion of the data and to document it in the conceptual schema. Then external and internal schemas for those data are built and mapped to the conceptual schema. Subsequently, other portions of the data resource are incorporated into the conceptual schema, and new external and internal schemas and mappings are developed. The CDM is populated gradually, in increments, rather than all at once. It evolves through time.

A conceptual schema is represented by a semantic data model. The IISS uses the IDEF1 methodology, with certain extensions from DACOM's Data Modeling Technique. (Subsequent to the development of CDM subsystem, IDEF1 was formally extended. See Appendix B for

references.) The data model reflects business policy, provides a rigorous view of the meaning of the data resource, and is independent of the physical implementation of the data resource.

Building a data model is a rigorous procedure, whose objective is to discover and document the semantic data structure in its most fundamental terms. The modeling is a multi-step process that requires substantial input from users who are expert in the subject area.

The fundamental steps of the CDM Integration Methodology are as follows:

1. Identify the scope of the initial increment of the conceptual schema.
2. Develop the data model for that initial increment of the conceptual schema.
3. Load the data model into the CDM database.
4. Identify any physical databases or files within the scope of data in the conceptual schema.
5. Load their internal schemas into the CDM database.
6. Build the conceptual-to-internal schema mappings for the internal schemas loaded in Step 5.
7. Load the conceptual-to-internal schema mappings into the CDM database.
8. Determine which users/application programs should have external schemas mapped from the conceptual schema.
9. Design the external schemas identified in Step 8, and their mappings to the conceptual schema.
10. Load the external schemas and external-to-conceptual schema mappings into the CDM database.
11. Identify the scope of the next increment to the conceptual schema.
12. Develop the data model for the next increment of the conceptual schema.
13. Integrate the data model from Step 12 with the data model of the existing conceptual schema.
14. Load the integrated data model into the CDM database.
15. Verify that the conceptual-to-internal and external-to-conceptual schema mappings are still valid, correcting them as needed.
16. Identify any additional physical databases or files that are now within the scope of the extended conceptual schema.

17. Load their internal schemas into the CDM database.
18. Build the conceptual-to-internal schema mappings for the incremented portions of the conceptual schema.
19. Load the conceptual-to-internal schema mappings into the CDM database.
20. Identify any additional users or application programs that should be supported by the extended conceptual schema.
21. Design external schemas to support the users/application programs identified in Step 20, and develop their external-to-conceptual schema mappings.
22. Load the external schemas and external-to-conceptual schema mappings from Step 21 into the CDM database.
23. Repeat Steps 11 through 22 for each increment to the conceptual schema.

The evolutionary strategy for the conceptual schema should be developed early in the life of the above cycle. The strategy should ensure that the common data resource evolves in a manner that serves the enterprise's need for controlled, shared data. One tactic is to define the initial scope by that of an existing database that has a corresponding data model. Ideally, that database would contain core information of high interest to the target user community.

Perhaps the most important point to understand about the CDM Integration Methodology is that the incorporation of additional data into the common data resource MUST be done in conjunction with the existing conceptual schema. No data can be accessed using the CDM integrated facilities, including the Neutral Data Manipulation Language, unless they are known to the CDM. Adding data causes the conceptual schema to expand in a consistent manner that enables integration to occur. By contrast, adding data to an environment that does not use conceptual schema technology just adds more fragmentation to what is probably already at best an interfaced (not integrated) system.

Applying the CDM Integration Methodology is not like swallowing a pill. It requires precise knowledge of the meanings of the data that are to be available in the integrated common data resource. It means not just building IDEF1 models for those databases, but also analyzing the models for overlap, synonyms, homonyms, and all the incipient anomalies and quirks that somehow have crept into our database structures over the years. The cost is measured in man-months of effort; the benefits are integration and a knowledge base that can be built on and evolved in the future.

2.1.4 Contributions to IIRIASSPA

The use of the Common Data Model and the Three-Schema Architecture allows an organization to benefit from contributions to IIRIASSPA, which are part of the objectives of the USA's Integrated Computer Aided Manufacturing (ICAM) project to develop the Integrated Information Support System (IISS). The contributions can best be summarized as follows:

Independence - the IISS allows the separation of the description and manipulation of logical data structures from the actual physical data representations and isolates implementation changes from user views and programs.

Relatability - the NDDL used in building the CDM allows the CDM Administrator to define, modify, and maintain relationships among data.

Resiliency/Recoverability - although not specifically addressed by the CDM, the design of the CDM Processor provides the ability to recover from failures without damage to the data resource.

Integrity - is provided through the use of data integrity constraints, which the application may specify and the CDM Processor enforces.

Accessibility - the NDDL allows the definition of data that resides not only in different databases but also on different computers.

Security - not expressly addressed by the CDM.

Shareability - is provided by support of multiple user views (i.e., external schemas) of the data resource.

Performance - the NDML, by use of the CDM, allows data [B from multiple resources to be addressed in a cost-effective manner in a distributed environment.

Administration - by providing a means of documenting the meanings in the data resource and of providing a vehicle by which consistency can be maintained even as the scope of the CDM is extended. It also allows the maintenance of information about data in different databases.

2.2 Basic Components of the Design

The Common Data Model(CDM) subsystem is comprised of three components:

1. The CDM database, which is the database dictionary of the IISS
2. A logical model of the CDM database called CDM
3. The CDM Processor (CDMP), which is the distributed database manager of the IISS

This section will briefly discuss each of these basic components and show how they interrelate, one with another.

2.2.1 The CDM Database

The CDM database is the database dictionary of the IISS. It captures knowledge of the locations, characteristics, and interrelationships of all shared data in the system. The most significant feature of the CDM database is that it implements the ANSI/X3/SPARC concepts of the three-schema approach to data management. These three types of schemas are the conceptual schema (CS), the internal schemas (IS), and the external schemas (ES).

The conceptual schema describes a neutral, integrated view of the shared data resource. There is one conceptual schema in an enterprise. It is independent of physical database structures and boundaries and is neutral to biases of individual applications. Each external schema represents a user or application view of data. Requests are made against external schemas. Each internal schema represents an external schema to the local DBMS.

The CDM database is implemented as a relational database, which presently resides on a VAX 11/780 computer. It is accessed by the CDMP at compile-time to generate appropriate local DBMS calls against internal schemas to process a user's NDML request against an external schema.

The CDM database is represented logically using a semantic data modeling technique called IDEF1. This method of data modeling is a hybrid of the entity-relationship approach, the relational model, and the Smith's 2D data abstraction approach. This logical model of the CDM database is called CDM1.

2.2.2 CDM1

CDM1 is a model of metadata, i.e., data about data. It gives the logical structure of the CDM database which maintains the metadata. These metadata describe the meanings and characteristics of user data.

The conceptual schema portion of the CDM1 model is related to portions that describe internal and external schemas. An internal schema describes a local database structure in just enough detail to give the CDMP adequate information to generate code that can be processed by the pertinent local DBMS. Because one of the requirements of the IISS is that it provide integration of data in existing databases, the mappings between the conceptual schema metadata and the internal schema metadata are not simple. IISS does not have the luxury of supporting only certain clean database structures. It is very likely that an attribute may be represented by one or more data files, which may be in different databases and even on different computers, or by relationships between record types.

An external schema describes the portion of the conceptual schema that is within the purview of a user or application. An external schema is equivalent to a view in the relational model.

The conceptual-to-external schema mapping part of the CDM1 is straightforward. The present implementation of the CDM subsystem supports any external schema that can be formed by joining conceptual schema entities and selecting attributes.

Thus, the CDM1 model is a semantic data model that describes the logical structure of the CDM database. The CDM1 represents the conceptual schema, the internal schemas and their mappings from the conceptual schema, and the external schemas and their mappings from the conceptual schema.

2.2.3 The CDM Processor

The CDMP is the distributed database manager of the IISS. It builds on top of local DBMS services to provide data access. The CDMP plays both a compile-time and a run-time role in the processing of transactions. The compile-time component is called the CDMP Precompiler. The run-time components are called the CDMP Distributed Request Supervisor (DRS) and the CDMP Aggregator.

2.2.3.1 CDMP Precompiler

The CDMP Precompiler performs the following functions for each data request:

1. Parses the request
2. Transforms the request from an external schema access to a conceptual schema access
3. Decomposes the request into subrequests, each of which accesses one internal schema
4. Determines an appropriate access path for each subrequest generating code that can be processed by the pertinent local DBMS
5. Generates code to transform any data to be extracted from local databases from internal to conceptual schema format (this code is called a Request Processor or RP)
6. Generates code to transform any data results from conceptual to external schema format and to perform statistical calculations (this code is called a C/E Transformer or CEX)
7. Generates code to invoke appropriate RPs and CEXs at run-time, via calls to the NTM Subsystem

The CDMP Precompiler accesses the CDM database to find metadata for the inter-schema transforms and integrity constraints for update requests.

After successful precompilation of a user's program, which contains embedded data requests in a SQL-like language called the Neutral Definition/Manipulation Language (NDML), the CDMP has produced the following code modules:

1. Modified user program will activate appropriate processes (RP's and CEX's) at runtime.
2. One Request Processor (RP) per DBMS that manages data to be accessed by the user program.
3. One Conceptual-to-External Transformer (CEX), which will deliver query results to the modified user program at run-time.

2.2.3.2 Distributed Request Supervisor

There are presently two CDMP Distributed Request Supervisor (DRS), one residing on the IBM node, the other on the VAX which have responsibility for scheduling and coordinating the various subrequests of user transactions. The DRS uses request graphs produced by the CDMP Precompiler to determine which operations are to be performed where. The DRS also uses knowledge of communications costs and intermediate result volumes in its algorithm for scheduling RPs.

Request Processors always deliver results as relations. The relations are operated upon by the Aggregators.

2.2.3.3 Aggregators

An Aggregator is called to perform a single function; for example, a union or a join, or an outer join on two sets of data, each of which exists in a single sequential file. These data sets are the results of an RPP or another Aggregator.

An Aggregator always deals with data in conceptual schema format.

SECTION 3

RESPONSIBILITIES OF THE CDM ADMINISTRATOR

The role that the CDM Administrator plays in the IISS environment is not unlike that of the database administrator in that the CDMA is responsible for the following:

1. Establishing Data Standards
2. Maintaining the CDM
3. Protecting the CDM
4. Facilitating Use of the CDM

Each of these areas is of major importance to the organization and a failure to properly administer either of these areas of responsibility can cost the organization dearly.

3.1 Establishing Data Standards

One of the early roles of the CDMA is the establishment of data standards. Part of this work has already been initiated during the development of the CDM1. The work that remains is to determine what types of standards to implement and to gain acceptance for the use of these standards. It should be noted that, without acceptable standards, it will be difficult, if not impossible, for the CDMA to enforce any level of standardization.

3.2 Maintaining the CDM

The CDMA must maintain the CDM. This entails the building of the initial conceptual schema (CS), internal schemas (IS), CS to IS mappings, external schemas (ES), and ES to CS mappings, as well as extending the model and modifying and deleting elements as needed. It is to be expected that the need for extending and modifying the CDM will grow over time, slowly at first, then growing rapidly as the benefits of the concept are proved before leveling off after several years.

3.3 Protecting the CDM

One of the most important responsibilities of the CDMA is the protection of the CDM against loss, theft, and corruption, be it intentional or not. At issue is the substantial investment that went into the development of the CDM and the potential damage that can be caused to the enterprise should the data fall into the wrong hands.

3.4 Facilitating Use of the CDM

The CDMA must make the CDM available to all those who can potentially gain from the use of the CDM and have legitimate reason to do so. This may involve making the CDM available on other computers in the network. It also involves communicating

UM 620341001
30 September 1990

with the CDM user and potential users as to the contents and performance of the CDM, as well as the usability of the data. Part of this communication will involve solving problems and answering questions and reporting the status of the CDM.

SECTION 4

MAINTAINING THE CONCEPTUAL SCHEMA

4.1 Methodology Overview

This section and its subsections (4.2 - 4.3) introduce the methodology for building and updating a conceptual schema. The portion of the CDM database that contains a conceptual schema is described, and the basic approach to developing a conceptual schema is presented. Detailed instructions for filling out the modeling forms are included.

4.1.1 CS Structure

A conceptual schema is essentially a single IDEF1 model that describes all of the common data in an enterprise. Consequently, its components are those of any IDEF1 model:

- Entity Classes
- Relation Classes
- Attribute Classes
- Attribute Use Classes
- Inherited Attribute Use Classes
- Key Classes
- Key Class Members

Detailed explanations of these can be found in the IDEF1 documentation. (Extensions to the IDEF1 language, referenced in Appendix C, simplify the IDEF1 terminology used here.)

In addition to the usual metadata (data about data) contained in any IDEF1 model, the conceptual schema requires certain new elements of metadata. Key class numbers are assigned to enable alternate key classes for the same entity class to be distinguished from one another. Tag numbers, tags (names), and tag labels are assigned to enable attribute use classes within the same entity class to be distinguished from one another. Data types and sizes are identified for all attribute classes.

The conceptual schema must conform to several rules that cause the data relationships and descriptions to be as explicit as possible. (Note: In these rules the phrase "any number" includes the possibility of zero.)

1. Single-Owner Rule: An entity class can own any number of attribute classes. Every attribute class is owned by exactly one entity class.
2. Every entity class contains one or more attribute use classes. Every attribute use class is contained in exactly one entity class.

3. Every attribute class appears as exactly one attribute use class in its owner entity class. An attribute class can also appear as any number of attribute use classes in any number of other entity classes. Every attribute use class corresponds to exactly one attribute class.
4. Every entity class has one or more key classes. Every key class is for exactly one entity class.
5. Every key class is composed of one or more key class members. Every key class member is in exactly one key class.
6. An attribute use class can be used as a member of any number of key classes for the entity class in which it is contained. An attribute use class cannot be used as more than one member of the same key class; i.e., every member of a key class must be a different attribute use class. An attribute use class in one entity class cannot be used as a member of a key class for any other entity class. Every key class member is exactly one attribute use class.
7. An entity class can be independent in any number of relation classes and dependent in any number. An entity class cannot be both independent and dependent in the same relation class. Every relation class has exactly two entity classes: one independent, one dependent.
8. A key class can migrate through any number of relation classes in which its entity class is independent. A key class cannot migrate through a relation class in which its entity class is dependent or one in which its entity class is not involved. Every relation class has exactly one key class from the independent entity class migrating through it into the dependent entity class.
9. Every relation class is a migration path for one or more inherited attribute use classes, one for each member of the key class that migrates through it. Every inherited attribute use class has exactly one relation class as its migration path.
10. Every member of the key class that migrates through a relation class creates exactly one inherited attribute use class in the dependent entity class for that relation class. Every inherited attribute use class is created from exactly one key class member.
11. Every attribute use class in an entity class represents either one attribute class that is owned by that entity class or one inherited attribute use class that migrated into that entity class. Every inherited attribute use class is represented by exactly one attribute use class.

12. Unique-Key Rule: No two entity instances in an entity class can have identical values in the samekey class for that entity class. For a multi-member key class, instances can have identical values for some members, but not for all.
13. No-Null Rule: Every entity instance in an entity class has a value in each attribute use class in that entity class.
14. No-Repeat Rule: No entity instance in an entity class can have more than one value in any attribute use class in that entity class. This rule is equivalent to the first normal form in the relational database model.
15. Full-Functional-Dependency Rule: No entity instance in an entity class can have a value in an owned, nonkey attribute use class that can be identified by less than the entire key value for that entity instance. This rule applies only to entity classes with multi-member key classes and is equivalent to the second normal form in the relational database model.
16. No-Transitive-Dependency Rule: No entity instance in an entity class can have a value in an owned, nonkey attribute use class that can be identified by the value in another owned or inherited, nonkey attribute use class in that entity class. This rule is equivalent to the third normal form in the relational database model.
17. Smallest-Key-Class Rule: No entity class with a multi-member key class can be split into two or more entity classes, each with fewer members in its key class, without losing some information. This rule is a combination and extension of the fourth and fifth normal forms in the relational database model.

4.1.2 Basic Approach (Onion Concept)

The complete conceptual schema for an enterprise contains thousands of entity classes and a corresponding number of relation classes, attribute classes, etc. It is much too large to be built all at once. Instead, it must be built in increments -- each one building on the prior ones, until the conceptual schema is complete. The increments are like the layers of an onion; as each layer is added, the onion gets a little larger.

The process of "growing" the conceptual schema involves two procedures, both of which are enhanced versions of the IDEF1 modeling procedure. The first is used to build the initial increment only. The second is used to build each additional increment. The only difference between the two is that the second must be concerned about the integration of the new increment with the existing conceptual schema. This involves being continually aware of which components of the conceptual schema are within the scope of the new increment and how any of those components will be affected by the addition of the new increment. These two procedures are in Sections 4.2 and 4.3, respectively.

4.1.3 Modeling Forms

Because the methodology for maintaining the conceptual schema is based on the IDEF1 information modeling methodology, it uses most of the IDEF1 forms:

| | |
|-------------------------------------|------------|
| Source Material Log | |
| Source Data List | |
| Entity Class Pool | |
| Entity Class Definition | |
| Relation Class Matrix | |
| Attribute Class Pool | |
| Kit Cover Sheet | |
| Entity Class Diagram | (optional) |
| Relation Class Definition | (optional) |
| Attribute Class Diagram | (optional) |
| Entity Class/Attribute Class Matrix | (optional) |
| Attribute Class Migration Index | (optional) |
| Author Page Control Log | (optional) |
| Index Control Log | (optional) |
| Kit Control Log | (optional) |
| Text Control Log | (optional) |
| FEO Control Log | (optional) |
| Entity Class Set Control Log | (optional) |
| Entity Class/Function View Matrix | (optional) |

Please refer to the IDEF1 documentation for detailed descriptions of these forms.

A few of the regular IDEF1 forms have certain shortcomings that make them unsuitable for use in directly loading the conceptual schema tables into the CDM database. The forms listed below were designed to eliminate those shortcomings:

- Relation Classes
- Owned Attribute Classes
- Inherited Attribute Classes

The rest of this section contains a detailed description and two samples (one blank, one filled in) of each of these forms.

NOTE: When using the NDDL (see Neutral Data Definition Language Users Guide, Pub. No. UM 620341100) for maintaining the conceptual schema in the CDM database, names should be substituted for any/all numbers on the modeling forms. A discussion of the NDDL can be found in Subsection 5.1.1.

Relation Classes Form

Purpose:

To provide a single source of information about relation classes that are to be described in the conceptual schema.

Instructions:

Fill in one or more pages for each entity class that is independent in a relation class. List only those relation classes in which the entity class is independent; do not list any relation classes in which it is dependent. Do not fill in a page for an entity class that is dependent in all of its relation classes.

| <u>Form Area</u> | <u>Explanation</u> |
|----------------------------------|--|
| 1. Independent Entity Class Name | Name of the entity class that is independent in the relation class. This will be the same for all relation classes entered on a page. It is included only to make the entry readable; it is not used in loading the conceptual schema. |
| 2. Relation Class Label | Label of the relation class. This is part of the unique identification of a relation class. |
| 3. R.C. Card. | Symbol for the cardinality of the relation class. |
| 4. Dependent Entity Class Name | Name of the entity class that is dependent in the relation class. It is included only to make the entry readable; it is not used in loading the conceptual schema. |
| 5. Dep. E.C. No. | Number of the entity class that is dependent in the relation class. |
| 6. Ind. K.C. No. | Number of the key class in the independent entity class that migrates through the relation class into the dependent entity class. |
| 7. Node | Number of the entity class that is independent in all of the relation classes listed on the page. |

All other form areas correspond to areas on the regular IDEF1 forms. Please refer to the IDEF1 documentation for details about those areas.

| | | | | | | |
|----------------------------------|----------------------------|--------------|--------------------------------|------------------|------------------|---------|
| USED AT | AUTHOR | DATE | WORKING | REVIEW | DATE | CONTEXT |
| | PROJECT | REV | DRAFT | | | |
| | NOTES 1 2 3 4 5 6 7 8 9 10 | | IN COMPLETED | | | |
| | | | THRU PATH | | | |
| Independent Entity Class Name | Relation Class Label | R.C. Card | Dependent Entity Class Name | Dep. E.C. No. | Ind. K.C. No. | |
| ① | ② | ③ | ④ | ⑤ | ⑥ | |
| MODE ⑦ | TITLE | | | Relation Classes | | NUMBER |

Figure 4-1. Relation Classes Form

| | | | | | | | | | |
|---------|----------------------------|------------------|------|----------|---|-------------|---------------|------|---------|
| USED AT | AUTHOR | DACOM (CEM, DRR) | DATE | Aug 1983 | X | WORKING | REAPPROPRIATE | DATE | CONTEXT |
| | PROJECT | 6201M MCMM | REV | | | FINAL | | | |
| | NOTES 1 2 3 4 5 6 7 8 9 10 | | | | | RECOMMENDED | | | |
| | | | | | | PUBLICATION | | | |

| Independent Entity Class Name | Relation Class Label | RC Card | Dependent Entity Class Name | Dep. E.C. No | Ind K.C. No |
|----------------------------------|-------------------------|------------|--------------------------------|-----------------|----------------|
| Op Exec Plan | Is | ← | OEP Group Member | E13 | K1 |
| Op Exec Plan | Has | ◇ | OEP Stored Item Req | E61 | K1 |
| Op Exec Plan | Is Used To Manufacture | ◆ | Op Exec Plan Part | E15 | K1 |
| Op Exec Plan | Is | ← | Op Exec Plan Comp | E14 | K1 |
| Op Exec Plan | Has | ◆ | Operation | E10 | K1 |
| Op Exec Plan | Has | ◇ | Op Exec Plan Obsl | E71 | K1 |

| | | | | | |
|------|-----|-------|------------------|--------|----|
| NODE | E11 | TITLE | Relation Classes | NUMBER | 62 |
|------|-----|-------|------------------|--------|----|

Figure 4-2. Relation Classes Form Example

Owned Attribute Classes Form

Purpose:

To provide a single source of information about owned attribute use classes that are to be described in the conceptual schema.

Instructions:

Fill in one or more pages for each entity class that owns an attribute use class, either key or nonkey. List only those attribute use classes that are owned by the entity class; do not list any attribute use classes that are inherited by the entity class. Do not fill in a page for an entity class that contains only inherited attribute use classes.

| <u>Form Area</u> | <u>Explanation</u> |
|----------------------|---|
| 1. Tag No. | Tag number for the attribute use class. |
| 2. A.C. Name & Label | Name, label, and any synonyms of the attribute use class. The name is listed first. The label is enclosed in parentheses and placed on the line below the name. If the name and label are identical, the label can be omitted. If the attribute use class has any synonyms, the term "Synonyms:" is placed below the name and label and the synonyms are listed under it. |
| 3. A.C. No. | Attribute class number for the attribute use class. |
| 4. A.C. Definition | Definition of the attribute use class. |
| 5. Type ID. | Format description for the attribute use class indicating data type (numeric, character, etc.), length, and decimal length (if applicable). The data type must be one from the CDM Data Type Table. |
| 6. Mbr. of K.C. No. | Number(s) of the key class(es) to which the attribute use class belongs, if any. |
| 7. Node | Number of the entity class that owns all of the attribute use classes listed on the page. |

All other form areas correspond to areas on the regular IDEF1 forms. Please refer to the IDEF1 documentation for details about those areas.

Inherited Attribute Classes Form

Purpose:

To provide a single source of information about inherited attribute use classes that are to be described in the conceptual schema.

Instructions:

Fill in one or more pages for each entity class that inherits an attribute use class. List only those attribute use classes that are inherited by the entity class; do not list any attribute use classes that are owned by the entity class. Do not fill in a page for an entity class that contains only owned attribute use classes.

| <u>Form Area</u> | <u>Explanation</u> |
|------------------|--|
| 1. Tag No. | Tag number for the attribute use class. |
| 2. Tag & Label | Name, label, and any synonyms of the attribute use class. The name is listed first. The label is enclosed in parentheses and placed on the line below the name. If the name and label are identical, the label can be omitted. If the attribute use class has any synonyms, the term "Synonyms:" is placed below the name and label, and the synonyms are listed under it. |
| 3. A.C. No. | Attribute class number for the attribute use class. |
| 4. Ind. E.C. No. | Number of the independent entity class from which the attribute use class was inherited. |
| 5. Ind. K.C. No. | Number of the key class in the independent entity class that migrated through the relation class named in the "Migration Path R.C. Label" area. |
| 6. Ind. Tag No. | Tag number of the attribute use class in the independent entity class that migrated to become this attribute use class. |

- | | | |
|----|------------------|---|
| 7. | Migration Path | Label of the relation class through which the attribute use class was inherited. |
| 8. | Mbr. of K.C. No. | Number(s) of the key class(es) to which the attribute use class belongs, if any. |
| 9. | Node | Number of the entity class that contains all of the attribute use classes listed on the page. |

All other form areas correspond to areas on the regular IDEF1 forms. Please refer to the IDEF1 documentation for details about those areas.

| | | | | | | |
|----------------------------|-------------------|-------------|--------------|--------|------|---------|
| USED AT | AUTHOR PROJECT | DATE REV | WORKING | ISSUED | DATE | CONTEXT |
| | | | DRAFT | | | |
| | | | RECOMMENDED | | | |
| | | | FINALIZATION | | | |
| NOTES 1 2 3 4 5 6 7 8 9 10 | | | | | | |

| Tag No | A C Name & Label | A C No. | A C. Definition | Type ID | Mbr. of K C. No |
|-----------|------------------|------------|-----------------|------------|--------------------|
| ① | ② | ③ | ④ | ⑤ | ⑥ |
| | | | | | |

| | | | | | |
|------|---|-------|-------------------------|--------|--|
| NODE | ⑦ | TITLE | Owned Attribute Classes | NUMBER | |
|------|---|-------|-------------------------|--------|--|

Figure 4-3. Owned Attribute Classes Form

| | | | | | | | | | | | | | |
|---------|--|-----------------------------|--|----------------|--|----------------|--|-----------|--|------|--|---------|--|
| USED AT | | AUTHOR: DACOM (CEM DRR) | | DATE: Aug 1983 | | X WORKING | | FR ADR II | | DATE | | CONTEXT | |
| | | PROJECT: 6201M MCMM | | REV | | DRAFT | | | | | | | |
| | | NOTES: 1 2 3 4 5 6 7 8 9 10 | | | | RECOMMENDED | | | | | | | |
| | | | | | | IMPLEMENTATION | | | | | | | |

| Tag No. | A.C. Name & Label | A.C. No. | A.C. Definition | Type ID. | Mbr of K.C. No. |
|---------|--|----------|--|----------|-----------------|
| T137 | Operation Execution Plan Group Identification (OEP Group ID) | A10 | A unique identifier assigned to identify groups of operation execution plans | N(4) | K01 |
| T134 | Status | A34 | A code that indicates where a group of operation execution plans is within its life cycle. | C(8) | |
| T135 | Total Operation Execution Plans (Total OEPs) | A35 | The total number of operation execution plans that make up the group | N(4) | |

| | | | | | |
|------|-----|-------|-------------------------|--------|----|
| NOTE | E12 | TITLE | Owned Attribute Classes | NUMBER | 69 |
|------|-----|-------|-------------------------|--------|----|

Figure 4-4. Owned Attribute Classes Form Example

| | | | | | | |
|---------|----------------------------|-------------|-------------|--------|------|---------|
| USED AT | AUTHOR | DATE REV | WORKING | REVIEW | DATE | CONTEXT |
| | PROJECT | | DRAFT | | | |
| | NOTES 1 2 3 4 5 6 7 8 9 10 | | RECOMMENDED | | | |
| | | | PUBLICATION | | | |

| Tag No. | Tag & Label | A.C. No. | Ind. E.C. No. | Ind. K.C. No. | Ind. Tag No. | Migration Path R.C. Label | Mbr. of K.C. No. |
|------------|-------------|-------------|------------------|------------------|-----------------|---------------------------|---------------------|
| ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ |
| | | | | | | | |

| | | | | | |
|------|---|-------|-----------------------------|--------|--|
| MODE | ⑨ | TITLE | Inherited Attribute Classes | NUMBER | |
|------|---|-------|-----------------------------|--------|--|

Figure 4-5. Inherited Attribute Classes Form

| | | | | | | | | |
|---------|----------------------------|------------------|------|--------------|---|---------|------|---------|
| USED AT | AUTHOR | DACOM (CEM, DRR) | DATE | Aug 1983 | X | WORKING | DATE | CONTEXT |
| | PROJECT | 6201M MCMM | REV | | | DRAFT | | |
| | NOTES 1 2 3 4 5 6 7 8 9 10 | | | RECOMMENDED | | | | |
| | | | | ILLUSTRATION | | | | |

| Tag No. | Tag & Label | A C No | Ind. E C No. | Ind. K C No. | Ind Tag No. | Migration Path R C Label | Mbr. of K C No |
|---------|--|--------|--------------|--------------|-------------|--------------------------|----------------|
| T73 | Requisition Number (Req No) | A09 | E20 | K01 | T28 | Is For | K01 |
| T191 | Issuing Manufacturing Area Identification (Iss Mtg Area ID) | A07 | E20 | K01 | T182 | Is For | K01 |
| T192 | Destination Manufacturing Area Identification (Dest Mtg Area ID) | A07 | E24 | K01 | T40 | Is | |

| | | | | | |
|------|-----|-------|-----------------------------|--------|-----|
| NODE | E67 | TITLE | Inherited Attribute Classes | NUMBER | 139 |
|------|-----|-------|-----------------------------|--------|-----|

Figure 4-6. Inherited Attribute Classes Form Example

4.2 Building the Initial CS

This section and its subsections (4.2.1 - 4.2.5) describe the procedure for initiating an enterprise's conceptual schema. The procedure is concerned with creating a detailed description (an information model) of a portion of the enterprise's common data and with collecting the data required to place that description in the CDM database as the first piece of the conceptual schema (the first layer of the onion). It is not concerned with deciding which portion of the common data to describe nor with setting up the CDM database and its utilities; these things must be done before starting the procedure. The procedure consists of six phases, the first five of which are patterned after those in IDEF1. The five IDEF1 phases are as follows:

- o Phase 0 - Starting the Project
- o Phase 1 - Defining Entity Classes
- o Phase 2 - Defining Relation Classes
- o Phase 3 - Defining Key Classes
- o Phase 4 - Defining Nonkey Attribute Classes

The procedure for the sixth phase, which consists of populating the CDM database with the conceptual schema, is described in Section 5. Each IDEF phase is described in a subsequent subsection.

4.2.1 Phase 0: Starting the Project

Objectives:

- o State the purpose, scope, and viewpoint for the information model.
- o Establish the project team.
- o Develop a phase-level project schedule.
- o Collect and catalog relevant source material.

This phase is patterned after Phase 0 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information.

Tasks:

1. The CDM Administrator appoints a project manager.
Usually, this will be the CDM Administrator.
2. The project manager states the purpose for building the information model.

This explains why the model is needed, i.e., what it will be used for. A model built with this procedure is primarily used to initiate the enterprise's conceptual schema. (It is not necessary to explain why the conceptual schema is needed.) If the model has other purposes, they should be mentioned also.

3. The project manager states the scope of the information model.

This sets the boundary of the model. It should be specific enough to be useful in deciding whether or not a particular element of common data should be included in the model. Some of the things that can be used as the basis for scoping a model are the following:

- o Information subjects: parts, employees, sales orders, etc.
- o Functions: engineering release, shop floor control, etc.
- o Existing computer files or databases
- o Existing computer application systems

4. The project manager states the viewpoints for the information model.

This explains the mental attitude or role that people should adopt when looking at and thinking about the model, i.e., in whose place they should put themselves. Usually, this will be the job title of someone who is intimately involved with the common data being modeled.

5. The project manager appoints the project team members.

The four roles to be filled are as follows:

- o Modeler - one or two IDEF1 experts.
- o Source - several subject experts, i.e., people who have in-depth knowledge about some or all of the common data being modeled.
- o Reviewer - several subject experts; some sources may also serve as reviewers. The CDM Administrator must also serve as a reviewer to ensure that the model, as it is developed, is properly documented for loading into the CDM database tables.
- o Librarian - a person who is trained and experienced in coordinating kit reviews and in maintaining files of model documentation; a modeler may also serve as the librarian.

6. The project manager appoints the acceptance review committee members.

This committee should consist of subject experts from the area being modeled and from other, related areas.

7. The project manager schedules the project phases.

Estimate the amount of effort needed to complete each phase (usually in man-weeks or man-months) and then convert those estimates to elapsed times and milestones based on the availability of the project team members. At this point, only the phases are scheduled; the individual tasks within a phase will be scheduled when that phase is started.

8. The project manager schedules the remaining Phase 0 tasks.

Estimate the amount of effort needed to perform each remaining task in this phase (usually in man-hours or man-days) and then convert those estimates to elapsed times and milestones based on the availability of the project team members who will perform those tasks. The schedules for the subsequent phases should be adjusted if they are inconsistent with these task schedules.

9. The modeler develops a data collection plan.

Determine what kinds of source material are needed and where and how to get that material.

10. The project manager conducts a project kick-off meeting attended by the project team members.

The objectives of the meeting are as follows:

- o To introduce the team members to one another and to the roles they will be performing.
- o To determine which members need IDEF1 training.
- o To present, discuss, and finalize the statements of purpose, scope, and viewpoint.
- o To present and discuss the project schedule.
- o To present, discuss, and finalize the data collection plan.

11. The modeler collects source material from the sources.

Gather the documents, policies, procedures, database designs, etc., and interview the sources in accordance with the data collection plan (Task 9).

12. The modeler catalogs the source material.

Prepare Source Material Log Forms and Source Data List Forms. If a database design is among the source material, the record names and data field names should be included in the source data list.

13. The modeler explains any author conventions.

These are deviations from or additions to the regular IDEF1 methodology. Mention the use of the three specially designed modeling forms: Relation Classes Form, Owned Attribute Classes Form, and Inherited Attribute Classes Form.

Deviation from IDEF1:

Usually, kits are not used to accomplish the review of the Phase 0 model documentation; the essentials are reviewed during the kick-off meeting (Task 10). However, the project manager may require that kits be used to supplement or replace the kick-off meeting.

4.2.2 Phase 1: Defining Entity Classes

Objective:

- o Identify and define the apparent entity classes that are within the scope of the model.

This phase is patterned after Phase 1 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information.

Tasks:

1. The project manager decides what method to use to review the Phase 1 model.

The options are to distribute review kits, to hold a walk-through meeting, or to do both. The factors to consider are the following:

- o Some team members may have to travel to attend a walk-through. How many trips can the project budget afford?
- o A review can usually be accomplished faster with a walk-through than with kits. Is there enough time to circulate kits, perhaps two or three times?
- o Some reviewers may have very limited time to spend on the project. How can their time be used most effectively, by reviewing a kit or by attending a walk-through? Will they devote time to reviewing a kit on their own?

2. The project manager schedules the Phase 1 tasks.

Estimate the amount of effort needed to perform each task in this phase (usually in man-hours or man-days) and then convert those estimates to elapsed times and milestones based on the availability of the project team members who will perform those tasks. The schedules for the subsequent phases should be adjusted if they are inconsistent with these task schedules.

3. The modeler builds an entity class pool.

Examine the entries in the source data list and deduce what sort of thing each entry identifies, describes, refers to, etc. For example:

- o Employee number, name, birth date, and salary are data elements about an employee; hence, an "Employee" entity class.
- o Part number, description, and dimensions are all about a part; hence, a "Part" entity class.

Each sort of thing is represented by an entity class. Talk to the sources when additional information is needed. The entity instances within an entity class should be distinguishable from one another by some unique identifier. Assign an entity class number to each entity class, and record it on an Entity Class Pool Form.

When examining record names from a database design, be careful to think about the "real-world thing" that each kind of record represents. Realize that several kinds of records may represent the same thing or, conversely, that one kind of record may represent several different things. Also, realize that certain kinds of records may be present for technical reasons only (performance, backup/recovery, etc.). Such records do not represent "real-world things" and should not result in entity classes being added to the pool.

4. The modeler defines each entity class.

Fill out an Entity Class Definition Form for each entity class in the pool. Talk to the sources when additional information about an entity class is needed. Check off each pool entry as it is dealt with.

Watch for synonyms (different names for the same thing) and homonyms (same name for different things). When there are synonyms for something, there is only one entity class to define. Use the most commonly used name as the "official" entity class name, and record it and the corresponding entity class number on an Entity Class Definition Form. Record the other names as synonyms on the form. In the pool, add a note to each synonym entry referring to the official name or number.

For a homonym, there are two or more entity classes to define, one for each thing that the term represents. Pick a new name for each thing to clarify the differences. Record the new names in the entity class pool along with a new entity class number for each, and fill out Entity Class Definition Forms. For example, if an order can be either something received by an enterprise from a customer, or something sent by an enterprise to a vendor, call the first a sales order and the second a purchase order, and fill out two definition forms.

5. The modeler, reviewers, and librarian participate in reviewing the Phase 1 model.

The method of review was selected in Task 1. The modelers prepare the review materials (kits or walk-through handouts), the reviewers read and comment on the materials, and the modelers respond to the comments. If kits are used, the librarian coordinates their circulation. The CDM Administrator reviews the model to ensure that all model documents are prepared properly for loading the CDM database tables.

4.2.3 Phase 2: Defining Relation Classes

Objective:

- o Identify and define the apparent relation classes that are within the scope of the model.

This phase is patterned after Phase 2 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information.

Tasks:

1. The project manager decides what method to use to review the Phase 2 model.

See Phase 1, Task 1, for the options and factors to consider.

2. The project manager schedules the Phase 2 tasks.

See Phase 1, Task 2, for details.

3. The modeler builds a relation class matrix.

List all of the entity classes across the top and down the left side of Relation Class Matrix Forms or on a large sheet of grid paper; the matrix is easier to work with when it is all on one sheet of paper. Then, determine which pairs of entity classes are related to each other. Look for data about one thing that is also data about another. For example:

- o Customer and Sales Order

A sales order has some data about the customer that placed it, such as customer number, name, address, etc.

- o Part and Purchase Order

A purchase order contains some data about the parts being ordered, such as part numbers, descriptions, dimensions, etc.

- o Department and Employee

One element of data about an employee is the department to which he/she is assigned, such as department number, name, etc.

- o Manufacturing Order and Employee

A manufacturing order has some data about the employees who performed its operations, such as employee numbers, names, etc.

Such sharing of data implies a relationship of some sort. Talk to the sources when additional information about such sharing of data is needed. If a database design is among the source material, the relationships it depicts may be useful. Place an "X" in the matrix at the intersection of each pair of related entity classes.

4. The modeler prepares overview diagrams (FEOs).

These diagrams are intended to show all of the entity and relation classes on just a few pages. Reviewers can usually understand overview diagrams better than individual entity class diagrams, so they will be the primary (or sole) depiction of the model. Each diagram should focus on a particular subject with which the reviewers will be comfortable (e.g., major activities), and each should contain about 10-to-20 entity classes and their relation classes. Use large sheets of paper (e.g., 11x17) and photo-reduction, if necessary.

Every entity and relation class in the matrix must appear in at least one diagram. Use some author convention to signify the entity classes that appear in more than one diagram (e.g., by broadening or double-lining the entity class boxes) and to identify which other diagrams they are in (e.g., by listing the diagram numbers near the entity class boxes). For example, if entity class E27 is in diagrams F1, F3, and F4:

- o List F3 and F4 near E27's box on F1.
- o List F1 and F4 near E27's box on F3.
- o List F1 and F3 near E27's box on F4.

Add the appropriate cardinality and a meaningful label to each relation class as it is drawn in a diagram. Talk to the sources when additional information about a relation class label and cardinality is needed. Cardinalities may be either specific or nonspecific; derived entity classes should not be introduced yet to avoid getting ahead of the reviewers. Check off each relation class in the matrix as it is drawn in a diagram (e.g., by circling the X in the matrix).

5. The modeler defines any additional entity classes that are introduced during this phase.

Whenever a new entity class is introduced, immediately document it by performing the tasks in Phases 1 and 2 that are needed to:

- o Update the entity class pool.
- o Prepare an Entity Class Definition Form.
- o Update the relation class matrix if it has been started.
- o Update the overview diagrams if they have been started.

6. The modeler, reviewers, and librarian participate in reviewing the Phase 2 model.

See Phase 1, Task 5 for details.
Deviation from IDEF1:

Usually, individual entity class diagrams are not prepared because the overview diagrams are easier to understand and review, and Relation Class Definition Forms are not filled out because the relation class labels are supposed to be self-descriptive. Also, the Related Entity Class Node Cross-Reference Form is replaced by the specially designed Relation.

Classes Form, which is called for in Phase 3. However, the project manager may require the use of any or all of these to supplement the model documentation called for above.

4.2.4 Phase 3: Defining Key Classes

Objectives:

- o Refine all nonspecific relation classes in the model.
- o Identify the apparent attribute classes that are within the scope of the model.
- o Identify and define a key class for each entity class in the model.
- o Validate every relation class in the model via key class migration.

This phase is patterned after Phase 3 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for

further information. Also, please refer to Subsection 5.2.2.1 for details on how to fill out the Relation Classes, Owned Attribute Classes, and Inherited Attribute Classes Forms.

Tasks:

1. The project manager decides what method to use to review the Phase 3 model.

See Phase 1, Task 1, for the options and factors to consider.

2. The project manager schedules the Phase 3 tasks.

See Phase 1, Task 2, for details.

3. The modeler refines the nonspecific relation classes.

Introduce a derived entity class for each nonspecific relation class and convert that relation class to a pair of specific relation classes as shown in Figure 4-7 at the end of this section. Assign entity class numbers to the derived entity classes, record them in the entity class pool, and fill out Entity Class Definition Forms. The sources may be able to recommend appropriate names and definitions for some derived entity classes.

Remove the nonspecific relation classes from the relation class matrix and the overview diagrams. Add the derived entity classes and the specific relation classes to the matrix and the diagrams. Retain the same focus for each diagram unless the reviewers suggested a change.

Also, update any optional documents that are affected.

4. The modeler eliminates any unneeded triads or other dual-path structures.

A dual-path structure is one composed of two or more related entity classes in which:

- o There are two paths connecting one entity class to another
- o One path is a single relation class
- o The other path is a series of relation classes (unless the structure has only two entity classes in which case the second path is a single relation class also)

See the examples in Figure 4-8 at the end of this section. Talk to the sources to determine whether the two paths are equal, unequal, or indeterminant. The paths are equal if, for each dependent entity instance, they both lead to the same independent entity instance. The paths are unequal if, for each dependent entity instance, they each lead to a different independent entity instance. The paths are indeterminant if they are

equal for some dependent entity instances and unequal for others. If the paths are equal, the single-relation-class path is redundant and must be removed from the relation class matrix and the overview diagrams (and from any optional documents in which appears).

5. The modeler fills out Relation Class Forms.

Record each relation class on a Relation Classes Form. Leave the Ind. K.C. No. column blank for now. As each relation class is recorded on a form, check it off on a copy of each overview diagram in which it appears (e.g., by circling the relation class labels).

6. The modeler builds an attribute class pool.

Examine the entries in the source data list and deduce what sort of characteristic each represents, where a characteristic is a data element that identifies, describes, refers to, etc., a thing being modeled. Each sort of characteristic is represented by an attribute class. Talk to the sources when additional information is needed. Assign an attribute class number to each attribute class, and record it on an Attribute Class Pool Form.

When examining data field names from a database design, realize that several data fields may represent the same kind of "real-world characteristic" or, conversely, that one data field may represent several different characteristics. For example:

- o SALES-ORDER-CUSTOMER-NUMBER, INVOICE-CUSTOMER-NUMBER, and ACCOUNTS-RECEIVABLE-CUSTOMER-NUMBER all represent the same characteristic of a customer, i.e., customer number.
- o SALESMAN-ASSIGNMENT-CODE may represent both the territory and the product for which the salesman is responsible.

Also, realize that certain data fields may be present for technical reasons only (e.g., record codes) and should not be included in the attribute class pool.

7. The modeler defines the key classes of the totally independent entity classes.

A totally independent entity class is one that is not dependent in any relation class. Select any one and find the attribute classes in the pool that make up its key class. Watch for attribute class synonyms and homonyms, and handle them like those for entity classes (Phase 1, Task 4). A few totally independent entity classes have two or more alternate key classes (e.g., employees can be uniquely identified by either employee numbers or Social

Security Numbers). Be sure to identify all key classes for such an entity class. Also, be sure each key class conforms to the following rules:

- o Single-Owned Rule
- o Unique-Key Rule
- o No-Null Rule
- o No-Repeat Rule
- o Smallest-Key-Class-Rule

See Section 4.1.1 for explanations of these rules. Define any new entity and relation classes needed to resolve rule violations. See Tasks 11 and 12 for details. Talk to the sources when additional information about a key class is needed.

Assign a key class number to each key class of the entity class (K1 for the first; K2 for the second, if any, etc.) and a tag number to each key class member. Fill out an Owned Attribute Classes Form, and record the key classes in the overview diagrams. Check off each attribute class in the pool as it is used.

8. The modeler migrates the key classes of the totally independent entity classes.

One of the key classes of the entity class from Task 7 must migrate through every relation class in which the entity class is independent. If it has two or more alternate key classes, only one can migrate through each relation class. The same one need not migrate through all of them however; one can migrate through some, another through others. The sources should be able to indicate which key class to use for each relation class. Record the number of the key class that migrates through a relation class in the Ind. K.C. No. column of the Relation Classes Form from Task 7.

Each member of the key class that migrates through a relation class becomes an inherited attribute class in the entity class that is dependent in that relation class. Fill out an Inherited Attribute Classes Form for each dependent entity class, i.e., those listed in the Dep. E.C. No. and Name columns of the Relation Classes Form. Record each inherited attribute class as follows:

- o Tag No. column: Assign a new tag number to each inherited attribute class.
- o Tag and Label column: Use the name and label of the key class member except in the following two situations:

- o If the key class member migrates through two relation classes into the same dependent entity class, it will appear as two inherited attribute classes, each of which must have a distinct name and label within the entity class. In this case, assign a new name and label to each. See Figure 4-9 at the end of this section for an example.
- o If a new name and label would be more descriptive, they may be used.
- o A.C. No. column: Use the attribute class number of the key class member even if a new name and label were assigned.
- o Ind. E.C. No. column: Use the number of the entity class that the key class member migrated from.
- o Ind. K.C. No. column: Use the key class number of the key class member.
- o Ind. Tag No. column: Use the tag number of the key class member.
- o Migration Path R.C. Label column: Use the label of the relation through which the key class member migrated.
- o Mbr. of K.C. No. column: Leave blank for now.

On copies of the overview diagrams, keep track of which relation classes have been used for key class migration (e.g., by circling the relation class labels).

Repeat Tasks 7 and 8 for each totally independent entity class.

9. The modeler defines the key classes of the remaining entity classes.

The remaining entity classes are those that are not totally independent, i.e., those that are dependent in at least one relation class. Key classes have migrated through some relation classes to appear as inherited attribute classes in some of these entity classes. Some have received all of their inherited attribute classes; others have not. One way to determine whether an entity class has is to examine the copies of the overview diagrams that were used to keep track of key class migration in Task 8. If each relation class in which the entity class is dependent has been used for key class migration, then the entity class has received all of its inherited attribute classes; otherwise, it has not.

Select any one entity class that has received all of its inherited attribute classes, and define its key class(es). The members of its key class(es) may include some of its inherited attribute classes or some new attribute classes from the pool or both. See Figure 4-10 at the end of this section for guidelines. Handle any synonyms and homonyms in the attribute class pool in the same way as those for entity classes (Phase 1, Task 4). Remember that the entity class may have two or more alternate key classes; be sure to identify all of them. Be sure each key class conforms to the following rules:

- o Single-Owner Rule
- o Unique-Key Rule
- o No-Null Rule
- o No-Repeat Rule
- o Smallest-Key-Class-Rule

See Subsection 4.1.1 for explanations of these rules. Define new entity and relation classes needed to resolve rule violations. See Tasks 11 and 12 for details. Talk to the sources when additional information about a key class is needed.

If a key class member comes from the attribute class pool, assign a tag number to it, check it off in the pool, and record it on an Owned Attribute Classes Form. Assign a key class number to each key class (K1 for the first; K2 for the second, if any, etc.), and record it in the Mbr. of K.C. No. column on the Owned Attribute Classes Form or the Inherited Attribute Classes Form where each key class member appears. If an attribute class, either owned or inherited, is a member of more than one key class, record the key class number of each. Also, record the key classes and any nonkey inherited attribute classes in the overview diagrams.

10. The modeler migrates the key classes of the remaining entity classes.

If the entity class from Task 9 is not independent in any relation classes, its key class does not migrate; see the last paragraph of this task. If it is independent in one or more relation classes, record the number of the key class that migrates through each one in the Ind. K.C. No. column of the Relation Classes Form. If the entity class has alternate key classes, record only one key class number for each relation class, although not all relation classes have to get the same number; the sources should be able to indicate which key class to use for each.

For each entity class that is listed in the Dep. E.C. No. and Name columns of the Relation Classes Form, fill out an Inherited Attribute Classes Form as described in Task 8. Also, as each relation class is used for key class migration, mark it on the overview diagram copies from Task 8.

Repeat Tasks 9 and 10 until key classes for all remaining entity classes have been defined and migrated.

11. The modeler defines any additional entity classes that are introduced during this phase.

Whenever a new entity class is introduced, immediately document it by performing the tasks in Phases 1 - 3 that are needed to:

- o Update the entity class pool.
- o Prepare an Entity Class Definition Form.
- o Update the relation class matrix.
- o Define the relation classes in which it is involved. See Task 12 for details.
- o Update the overview diagrams.
- o Define and migrate its key class(es) at the appropriate time during Tasks 7 - 10.
- o Update any optional documents that are affected.

12. The modeler defines any additional relation classes that are introduced during this phase.

Whenever a new relation class is introduced, immediately document it by performing the tasks in Phases 2 and 3 that are needed to:

- o Update the relation class matrix.
- o Update the overview diagrams.
- o Refine it if it is nonspecific.
- o Eliminate any unneeded dual-path structures.
- o Record it on a Relation Classes Form.
- o Validate it via key class migration at the appropriate time during Task 8 or 10.
- o Update any optional documents that are affected.

13. The modeler, reviewers, and librarian participate in reviewing the Phase 3 model.

See Phase 1, Task 5, for details.

Deviation from IDEF1:

The specially designed Relation Classes, Owned Attribute Classes, and Inherited Attribute Classes forms are used in place of the regular IDEF1 forms: Related Entity Class Node Cross-Reference, Attribute Class Definition (2), and Inherited Attribute Class Cross-Reference. The forms used are designed to facilitate loading the conceptual schema. Also, the following IDEF1 forms are not called for, but may be used at the discretion of the project manager:

- o Attribute Class Diagram
- o Entity Class/Attribute Class Matrix
- o Attribute Class Migration Index
- o Refinement Alternative Diagram
- o Entity Class/Function View Matrix

4.2.5. Phase 4: Defining Nonkey Attribute Classes

Objectives:

- o Identify and define the nonkey attribute classes that are within the scope of the model.
- o Identify the entity class that owns each nonkey attribute class.

This phase is patterned after Phase 4 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information. Also, please refer to Section 4.1.3 for details on how to fill out Owned Attribute Classes Forms.

Tasks:

1. The project manager decides what method to use to review the Phase 4 model.

See Phase 1, Task 1, for the options and factors to consider.
2. The project manager schedules the Phase 4 tasks.

See Phase 1, Task 2, for details.
3. The modeler populates the model with the nonkey attribute classes.

The nonkey attribute classes are those that were not used as members of any key classes in Phase 3, i.e., those that have not been checked off in the attribute class pool. Find the entity class that owns each of these according to the following rules:

- o Single-Owner Rule
- o No-Null Rule
- o Full-Functional-Dependency Rule
- o No-Transitive-Dependency Rule

See Section 4.1.1 for explanations of these rules. Define any new entity and relation classes needed to resolve any rule violations. See Tasks 4 and 5 for details. Talk to the sources when additional information about a nonkey attribute class is needed.

Assign a tag number to each nonkey attribute class, and record it on an Owned Attribute Classes Form. Check off each in the pool as it is used.

4. The modeler defines any additional entity classes that are introduced during this phase.

Whenever a new entity class is introduced, immediately document it by performing the tasks in Phases 1 - 3 that are needed to:

- o Update the entity class pool.
 - o Prepare an Entity Class Definition Form.
 - o Update the relation class matrix.
 - o Define the relation classes that it is involved in. See Task 5 for details.
 - o Update the overview diagrams.
 - o Define and migrate its key class(es).
 - o Update any optional documents that are affected.
5. The modeler defines any additional relation classes that are introduced during this phase.

Whenever a new relation class is introduced, immediately document it by performing the tasks in Phases 2 and 3 that are needed to:

- o Refine it if it is nonspecific.
 - o Eliminate any unneeded dual-path structures.
 - o Update the relation class matrix.
 - o Record it on a Relation Classes Form.
 - o Update the overview diagrams.
 - o Validate it via key class migration.
 - o Update any optional documents that are affected.
6. The modeler, reviewers, and librarian participate in reviewing the Phase 4 model.

See Phase 1, Task 5, for details.

Deviation from IDEF1:

The specially designed Owned Attribute Classes Form is used instead of the regular Attribute Class Definition Forms to facilitate loading the conceptual schema. Also, the following IDEF1 forms are not called for, but may be used at the discretion of the project manager:

- o Attribute Class Diagram
- o Entity Class/Attribute Class Matrix

See Subsection 5.2.1 for instructions on how to load.

4.3 Expanding the CS

This section and its subsections describe the procedure for expanding an enterprise's conceptual schema. The procedure is concerned with creating a detailed description (an information model) of a portion of the enterprise's common data, some or all of which is not already described in the conceptual schema, and with collecting the data required to place that description in the CDM database as an additional piece of the conceptual schema (another layer of the onion).

The procedures described in the following subsections correspond to the five IDEF1 phases discussed in the previous section.

4.3.1 Phase 0: Starting the Project

Objectives:

- o State the purpose, scope, and viewpoint for the information model.
- o Establish the project team.
- o Develop a phase-level project schedule.
- o Collect and catalog relevant source material.

This phase is patterned after Phase 0 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information. Also, please refer to Subsection 4.1.2 for details on how to fill out the Relation Classes, Owned Attribute Classes, and Inherited Attribute Classes forms.

Tasks:

1. The CDM Administrator appoints a project manager. Usually, this will be the CDM Administrator.
2. The project manager states the purpose for building the information model.
3. The project manager states the scope of the information model.

See Task 2 of Subsection 4.2.1.

See Task 3 of Subsection 4.2.1.

4. The project manager states the viewpoint for the information model.

See Task 4 of Subsection 4.2.1.

5. The project manager appoints the project team members.

See Task 5 of Subsection 4.2.1.

6. The project manager appoints the acceptance review committee members.

This committee should consist of subject experts from the area being modeled and from other, related areas.

7. The project manager schedules the project phases.

See Task 7 of Subsection 4.2.1.

8. The project manager schedules the remaining Phase 0 tasks.

See Task 8 of Subsection 4.2.1.

9. The modeler develops a data collection plan.

Determine what kinds of source material are needed and where and how to get that material.

10. The project manager conducts a project kick-off meeting attended by the project team members.

See Task 10 of Subsection 4.2.1.

11. The modeler collects source material from the sources.

Gather the documents, policies, procedures, database designs, etc., and interview the sources in accordance with the data collection plan (Task 9).

12. The modeler catalogs the source material.

Prepare Source Material Log Forms and Source Data List Forms. If a database design is among the source material, the record names and data field names should be included in the source data list.

13. The modeler examines the existing conceptual schema.

Identify the entity, relation, and attribute classes in the existing conceptual schema that appear to be within the scope of the model. Fill out the following forms from the descriptions in the conceptual schema:

- o Entity Class Definition Forms
- o Relation Classes Forms
- o Owned Attribute Classes Forms
- o Inherited Attribute Classes Forms
- o Relation Class Matrix Forms

To distinguish these elements of the conceptual schema from the new ones that will be documented during the course of this modeling project, prefix all of the identification numbers with the letter "C." For example:

- o Entity Class Number = CE12
- o Attribute Class Number = CA94
- o Tag Number = CT156
- o Key Class Number = CK1

14. The modeler explains any author conventions.

These are deviations from or additions to the regular IDEF1 methodology. Mention the use of the three specially designed modeling forms: Relation Classes Form, Owned Attribute Classes Form, and Inherited Attribute Classes Form. Also, explain that in order to distinguish between model elements that are already in the conceptual schema and those that are not, the identification numbers of the former will be prefixed with the letter "C" for conceptual while those of the latter will be prefixed with the letter "N.Y." for new.

Deviation from IDEF1:

Usually, kits are not used to accomplish the review of the Phase 0 model documentation; the essentials are reviewed during the kick-off meeting (Task 10). However, the project manager may require that kits be used to supplement or replace the kick-off meeting.

4.3.2 Phase 1: Defining Entity Classes

Objective:

- o Identify and define the apparent entity classes that are within the scope of the model.

This phase is patterned after Phase 1 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information.

Tasks:

1. The project manager decides what method to use to review the Phase 1 model.

See Task 1 of Subsection 4.2.2.

2. The project manager schedules the Phase 1 tasks.

See Task 2 of Subsection 4.2.2.

3. The modeler builds an entity class pool.

Examine the entries in the source data list and deduce what sort of thing each entry identifies, describes, refers to, etc. For example:

- o Employee number, name, birth date, and salary are data elements about an employee; hence, an "Employee" entity class.
- o Part number, description, and dimensions are all about a part; hence, a "Part" entity class.

Each sort of thing is represented by an entity class. Determine whether any of these entity classes are already in the conceptual schema and, if so, whether modeling forms were prepared for them in Phase 0, Task 13. Rely on the entity class definitions more than the names or labels in deciding whether a conceptual schema entity class represents the same sort of thing as an entity class deduced from the source data list. If any entity class is in the conceptual schema, but modeling forms were not prepared, prepare them now; see Phase 0, Task 13 for details. Talk to the sources when additional information is needed. The entity instances within an entity class should be distinguishable from one another by some unique identifier. Assign an entity class

number, prefixed with "N," to each new entity class, and record them on an Entity Class Pool Form. Do not record any conceptual schema entity classes in the pool.

When examining record names from a database design, be careful to think about the "real-world thing" that each kind of record represents. Realize that several kinds of records may represent the same thing or, conversely, that one kind of record may represent several different things. Also, realize that certain kinds of records may be present for technical reasons only (performance, backup/recovery, etc.). Such records do not represent "real-world things" and should not result in entity classes being added to the pool.

4. The modeler defines each entity class.

See Task 4 of Subsection 4.2.2.

Also, review the names, labels, and definitions of the conceptual schema entity classes, record any changes that are required on the Entity Class Definition Forms, and write "UPDATED" below the entity class number in the lower left corner.

5. The modeler, reviewers, and librarian participate in reviewing the Phase 1 model.

The method of review was selected in Task 1. The modelers prepare the review materials (kits or walk-through handouts), the reviewers read and comment on the materials, and the modelers respond to the comments. If kits are used, the librarian coordinates their circulation.

6. The CDM Administrator reviews the model to ensure that it is compatible with the conceptual schema.

Definitions are compared to see whether any entity, relation, or attribute classes that are identified as new in the model are really the same as those that are already in the conceptual schema, possibly with different names or labels. Also, each proposed conceptual schema update is evaluated to gauge its impact on the existing CS/ES and CS/IS mappings.

4.3.3. Phase 2: Defining Relation Classes

Objective:

- o Identify and define the apparent relation classes that are within the scope of the model.

This phase is patterned after Phase 2 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information.

Tasks:

1. The project manager decides what method to use to review the Phase 2 model.

See Phase 1, Task 1, for the options and factors to consider.

2. The project manager schedules the Phase 2 tasks.

See Phase 1, Task 2, for details.

3. The modeler builds a relation class matrix.

See Task 3 of Subsection 4.2.3.

4. The modeler prepares overview diagrams (FEOs).

See Task 4 of Subsection 4.2.3.

5. The modeler defines any additional entity classes that are introduced during this phase.

Whenever a new entity class is introduced, double-check the conceptual schema to see if it is already there. Rely on the entity class definitions more than the names or labels in deciding whether a conceptual schema entity class represents the same sort of thing as a new entity class. If a new entity class is already described in the conceptual schema, prepare the modeling forms listed in Phase 0, Task 13. If it is not, immediately document it by performing the tasks in Phases 1 and 2 that are needed to:

- o Update the entity class pool
- o Prepare an Entity Class Definition Form.
- o Update the relation class matrix if it has been started.
- o Update the overview diagrams if they have been started.

6. The modeler, reviewers, and librarian participate in reviewing the Phase 2 model.

See Task 5 of this section for details.

Deviation from IDEF1:

Usually, individual entity class diagrams are not prepared because the overview diagrams are easier to understand and review, and Relation Class Definition Forms are not filled out because the relation class labels are supposed to be self-descriptive. Also, the Related Entity Class Node Cross-Reference Form is replaced by the specially designed Relation Classes Form, which is called for

in Phase 3. However, the project manager may require the use of any or all of these to supplement the model documentation called for above.

4.3.4 Phase 3: Defining Key Classes

Objectives:

- o Refine all nonspecific relation classes in the model.
- o Identify the apparent attribute classes that are within the scope of the model.
- o Identify and define a key class for each entity class in the model.
- o Validate every relation class in the model via key class migration.

This phase is patterned after Phase 3 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information. Also, please refer to Section 4.1.3 for details on how to fill out the Relation Classes, Owned Attribute Classes, and Inherited Attribute Classes Forms.

Tasks:

1. The project manager decides what method to use to review the Phase 3 model.

See Task 1 of Subsection 4.2.1.

2. The project manager schedules the Phase 3 tasks.

See Task 2 of Subsection 4.2.1.

3. The modeler refines the nonspecific relation classes.

Introduce a derived entity class for each nonspecific relation class and convert that relation class to a pair of specific relation classes as shown in Figure 4-7 at the end of this section. Assign entity class numbers, prefixed with "N," to the derived entity classes, record them in the entity class pool, and fill out Entity Class Definition Forms. The sources may be able to recommend appropriate names and definitions for some derived entity classes.

Remove the nonspecific relation classes from the relation class matrix and the overview diagrams. Add the derived entity classes and the specific relation classes to the matrix and the diagrams. Retain the same focus for each diagram unless the reviewers suggested a change. Also, update any optional documents that are affected.

4. The modeler eliminates any unneeded triads or other dual-path structures.

A dual-path structure is one composed of two or more related entity classes in which:

- o There are two paths connecting one entity class to another
- o One path is a single relation class
- o The other path is a series of relation classes (unless the structure has only two entity classes in which case the second path is a single relation class also)

See the examples in Figure 4-8 at the end of this section. Talk to the sources to determine whether the two paths are equal, unequal, or indeterminant. The paths are equal if, for each dependent entity instance, they both lead to the same independent entity instance. The paths are unequal if, for each dependent entity instance, they each lead to a different independent entity instance. The paths are indeterminant if they are equal for some dependent entity instances and unequal for others. If the paths are equal, the single-relation-class path is redundant and must be removed from the model, i.e., from the relation class matrix and the overview diagrams (and from any optional documents in which it appears).

If the relation class that must be removed is already described in the conceptual schema, it should already be listed on a Relation Classes Form from Phase 0, Task 13. Write "DELETE" in the margin next to it and write "UPDATED" below the entity class number in the lower left corner.

If the dependent entity class in that relation class is from the conceptual schema, the inherited attribute classes that it received via key class migration through that relation class must be removed also. Write "DELETE" in the margin next to each one on the Inherited Attribute Classes Form, and write "UPDATED" below the entity class number in the lower left corner. If any of them is a key class member in the dependent entity class, that key class is now incomplete and must be removed; see Task 13 for details.

5. The modeler fills out Relation Class Forms.

See Task 5 of Subsection 4.2.4.

6. The modeler builds an attribute class pool.

Examine the entries in the source data list and deduce what sort of characteristic each represents, where a characteristic is a data element that identifies, describes, or refers to, a thing being modeled. Each sort of characteristic is represented by an at-

tribute class. Determine whether any of the attribute classes are already in the conceptual schema and, if so, whether modeling forms were prepared for them in Phase 0, Task 13. Rely on the attribute class definitions more than the names or labels in deciding whether a conceptual schema attribute class represents the same sort of characteristic as an attribute class deduced from the source data list. If an attribute class is in the conceptual schema, but modeling forms were not prepared, prepare them now; see Phase 0, Task 13, for details. Talk to the sources when additional information is needed. Assign an attribute class number, prefixed with "N," to each new characteristic deduced from the source data list, and record them on Attribute Class Pool Forms. When examining data field names from a database design, realize that several data fields may represent the same kind of "real-world characteristic" or, conversely, that one data field may represent several different characteristics. For example:

- o SALES-ORDER-CUSTOMER-NUMBER, INVOICE-CUSTOMER-NUMBER, and ACCOUNTS-RECEIVABLE-CUSTOMER-NUMBER all represent the same characteristic of a customer, i.e., customer number.
- o SALESMAN-ASSIGNMENT-CODE may represent both the territory and the product for which the salesman is responsible.

Also, realize that certain data fields may be present for technical reasons only (e.g., record codes) and should not be included in the attribute class pool.

7. The modeler defines the key classes of the totally independent entity classes.

A totally independent entity class is one that is not dependent in any relation classes. Select any one and find the attribute classes in the pool that make up its key class. If the entity class is already in the conceptual schema, at least one key class has already been defined for it. However, others may be discovered here because of new owned attribute classes. Watch for attribute class synonyms and homonyms, and handle them like those for entity classes (Phase 1, Task 4). A few totally independent entity classes have two or more alternate key classes (e.g., employees can be uniquely identified by either Social Security or employee numbers). Be sure to identify all key classes for such an entity class. Also, be sure each key class conforms to the following rules:

- o Single-Owned Rule
- o Unique-Key Rule
- o No-Null Rule
- o No-Repeat Rule
- o Smallest-Key-Class-Rule

See Subsection 4.1 for explanations of these rules. Define any new entity and relation classes needed to resolve rule violations. See Tasks 11 and 12 for details. If an attribute class that is needed as a key class member for a new entity class is already owned by a conceptual schema entity class, a relationship exists between those two entity classes. If it is not already documented as a new relation class, it must be before the key class of the new entity class can be defined; see Task 12 for details. If the new entity class is dependent in the new relation class, it is no longer totally independent, so its key class cannot be defined until Task 9. If the new entity class is independent in the relation class, the ownership of the attribute class must be changed; it is owned by the new entity class, not by the one in the conceptual schema. Record it on an Owned Attribute Classes Form for the new entity class, using the same name, label, definition, domain (type and size), and attribute class number, prefixed with "C," but assign a new tag number, prefixed with "N." Write "DELETE" in the margin next to the attribute class on the form for the conceptual schema entity class and write "UPDATED" below the entity class number in the lower left corner. If it is a key class member in the conceptual schema entity class, that key class is now incomplete and must be removed; see Task 13 for details. Talk to the sources when additional information about a key class is needed.

Assign a key class number, prefixed with "N," to each new key class of the entity class (NK1 for the first; NK2 for the second, if any, etc.). Assign a tag number, prefixed with "N," to each new attribute class that is a key class member; record it on an Owned Attribute Classes Form, and check it off in the attribute class pool. Record the key classes, both new ones and ones from the conceptual schema, in the overview diagrams.

Also, review the name, label, and definition of each conceptual schema attribute class that is a key class member; record any changes that are required on the Owned Attribute Classes Form where it appears, write "CHANGE" in the margin next to it, and write "UPDATED" below the entity class number in the lower left corner.

8. The modeler migrates the key classes of the totally independent entity class.

One of the key classes of the entity class from Task 7 must migrate through every relation class in which the entity class is independent. A key class has already migrated through every conceptual schema relation class, but some may have had that migration undone in Task 13, i.e., those with a circled key class number in the Ind. K.C. No. column of a Relation Classes Form and with "OMIT" written in the

margin. Only these and the new relation classes, i.e., those without a key class number in that column, need to be considered here. If the entity class has two or more alternate key classes, only one can migrate through each relation class. The same one need not migrate through all of them; however, one can migrate through some, another through others. The sources should be able to indicate which key class to use for each relation class. For a new relation class, record the key class number in the Ind. K.C. No. column of the Relation Classes Form from Task 7. For a conceptual schema relation class that is having its key class migration redone, if the key class number is the same as the one that is already in Ind. K.C. No. column, erase the circle around it and erase "OMIT" in the margin. If the key class numbers are different, replace the circled one with the new one and change "OMIT" to "CHANGE" in the margin.

Each member of the key class that migrates through a relation class becomes an inherited attribute class in the entity class that is dependent in that relation class. Fill out an Inherited Attribute Classes Form for each dependent entity class, i.e., those listed in the Dep. E.C. No. and Name columns of the Relation Classes Form. If the dependent entity class is already in the conceptual schema, use the Inherited Attribute Classes Form that was prepared in Phase 0, Task 13. Record each new inherited attribute class as follows:

- o Tag No. column: Assign a new tag number, prefixed with "N," to each inherited attribute class. If an inherited attribute class replaces an owned attribute class whose ownership was changed in Task 8 or 10, use the tag number prefixed with "C" that was assigned to that owned attribute class, and change "DELETE" to "NEW OWNER" in the margin next to that owned attribute class on the Owned Attribute Classes Form.
- o Tag and Label column: Use the name and label of the key class member except in the following two situations:
 - o If the key class member migrates through two relation classes into the same dependent entity class, it will appear as two inherited attribute classes, each of which must have a distinct name and label within the entity class. In this case, assign a new name and label to each. See Figure 4-9 at the end of this section for an example.
 - o If a new name and label would be more descriptive, they may be used.

- o A.C. No. column: Use the attribute class number of the key class member, even if a new name and label were assigned.
- o Ind. E.C. No. column: Use the number of the entity class from which the key class member migrated.
- o Ind. K.C. No. column: Use the key class number of the key class member.
- o Ind. Tag No. column: Use the tag number of the key class member.
- o Migration Path R.C. Label column: Use the label of the relation class through which the key class member migrated.
- o Mbr. of K.C. No. column: Leave blank for now.

If an inherited attribute class that was removed from a conceptual schema entity class in Task 4 or 13 is being re-established, do not record it as described above. Instead, reuse the one that is already recorded on the Inherited Attribute Classes Form. Erase "DELETE" from the margin. If any of the values in the following columns need to be changed, replace them with the new values and write "CHANGE" in the margin:

- o Tag and Label Column
- o Ind. E.C. No. Column
- o Ind. K.C. No. Column
- o Ind. Tag No. Column
- o Migration Path R.C. Label Column

If the Mbr. of K.C. No. column contains any key class numbers, circle each and write "OMIT" in the margin.

On copies of the overview diagrams, keep track of which relation classes have been used for key class migration, including those from the conceptual schema that had already been used (e.g., by circling the relation class labels).

Repeat Tasks 7 and 8 for each totally independent entity class.

9. The modeler defines the key classes of the remaining entity classes.

The remaining entity classes are those that are not totally independent, i.e., those that are dependent in at least one relation class. Key classes have migrated through some relation classes to appear as inherited attribute classes in some of these entity classes. Some have received all of their inherited attribute classes; others have not. One way to determine whether an entity class has is to examine

the copies of the overview diagrams that were used to keep track of key class migration in Task 8. If each relation class that the entity class is dependent in has been used for key class migration, then the entity class has received all of its inherited attribute classes; otherwise it has not. Select any one entity class that has received all of its inherited attribute classes, and define its key class(es). If the entity class is already in the conceptual schema, at least one key class has already been defined for it. However, if one was removed in Task 13, it must be re-established or a new one must be defined. Also, other key classes may be discovered here because of new owned or inherited attribute classes. The members of its key class(es) may include some of its inherited attribute classes or some of the new attribute classes from the pool or both. See Figure 4-10 at the end of this section for guidelines. Handle any synonyms and homonyms in the attribute class pool in the same way as those for entity classes (Phase 1, Task 4). Remember that the entity class may have two or more alternate key classes; be sure to identify all of them. Be sure each key class conforms to the following rules:

- o Single-Owner Rule
- o Unique-Key Rule
- o No-Null Rule
- o No-Repeat Rule
- o Smallest-Key-Class-Rule

See Subsection 4.1 for explanations of these rules. Define new entity and relation classes needed to resolve rule violations. See Tasks 11 and 12 for details. If an attribute class that is needed as a key class member for a new entity class is already owned by a conceptual schema entity class, a relationship exists between those two entity classes. If it is not already documented as a new relation class, it must be before the key class of the new entity class can be defined; see Task 12 for details. If the new entity class is dependent in the relation class, its key class cannot be defined until one from the independent entity class has migrated through the new relation class. If the new entity class is independent in the relation class, the ownership of the attribute class must be changed; it is owned by the new entity class, not by the one in the conceptual schema. Record it on an Owned Attribute Classes Form for the new entity class, using the same name, label, definition, domain (type and size), and attribute class number, prefixed with "C," but assign a new tag number, prefixed with "N." Write "DELETE" in the margin next to the attribute class on the form for the conceptual schema entity class and write "UPDATED" below the entity class number in the lower left corner. If it is a key class member in the conceptual schema entity class, that key class is now

incomplete and must be removed; see Task 13 for details. Talk to the sources when additional information about a key class is needed.

Assign a key class number, prefixed with "N," to each new key class (NK1 for the first; NK2 for the second, if any, etc.). If a key class that was removed in Task 13 is being re-established, reuse its original key class number, prefixed with "C." Assign a tag number, prefixed with "N," to each new key class member that comes from the attribute class pool, check it off in the pool, and record it on an Owned Attribute Classes Form.

Also, review the name, label, and definition of each conceptual schema attribute class that is a key class member, record any changes that are required on the Owned Attribute Classes Form where it appears, write "CHANGE" in the margin next to it, and write "UPDATED" below the entity class number in the lower left corner.

Identify each new key class member by recording its key class number in the Mbr. of K.C. No. column on either the Owned Attribute Classes Form or the Inherited Attribute Classes Form. If an attribute class, either owned or inherited, is a member of more than one key class, record the key class number of each. If an attribute class is being re-established as a member of a key class that was removed in Task 13, erase the circle around the key class number in the Mbr. of K.C. No. column of the Owned or Inherited Attribute Classes Form and erase "OMIT" from the margin. Also, record the key classes and any nonkey inherited attribute classes, both new ones and ones from the conceptual schema, in the overview diagrams.

10. The modeler migrates the key classes of the remaining entity classes.

If the entity class from Task 9 is not independent in any relation classes, its key class does not migrate; see the last paragraph of this task. If it is independent in one or more relation classes, one of its key classes must migrate through each. A key class has already migrated through every conceptual schema relation class, but some may have had that migration undone in Task 13, i.e., those with a circled key class number in the Ind. K.C. No. column of a Relation Classes Form and with "OMIT" written in the margin. Only these and the new relation classes, i.e., those without a key class number in that column, need to be considered here. Record the number of the key class that migrates through each new relation class in the Ind. K.C. No. column of the Relation Classes Form. If the entity class has alternate key classes, record only one key class number for each relation class, although not all

relation classes have to get the same number; the sources should be able to indicate which key class to use for each. For a conceptual schema relation class that is having its key class migration redone, if the key class number is the same as the one that is already in Ind. K.C. No. Column, erase the circle around it and erase "OMIT" in the margin. If the key class numbers are different, replace the circled one with the new one and change "OMIT" to "CHANGE" in the margin.

For each entity class that is listed in the Dep. E.C. No. and Name columns of the Relation Classes Form, fill out an Inherited Attribute Classes Form as described in Task 8. Also, keep track of which relation classes have been used for key class migration, including those from the conceptual schema, by marking them on the overview diagram copies from Task 8.

Repeat Tasks 9 and 10 until key classes for all remaining entity classes have been defined and migrated.

11. The modeler defines any additional entity classes that are introduced during this phase.

Whenever a new entity class is introduced, double-check the conceptual schema to see if it is already there. Rely on the entity class definitions more than the names or labels in deciding whether a conceptual schema entity class represents the same sort of thing as a new entity class. If a new entity class is already described in the conceptual schema, prepare the modeling forms listed in Phase 0, Task 13. If it is not, immediately document it by performing the tasks in Phases 1 - 3 that are needed to:

- o Update the entity class pool.
- o Prepare an Entity Class Definition Form.
- o Update the relation class matrix.
- o Define the relation classes in which it is involved. See Task 12 for details.
- o Update the overview diagrams.
- o Define and migrate its key class(es) at the appropriate time during Tasks 7 - 10.
- o Update any optional documents that are affected.

12. The modeler defines any additional relation classes that are introduced during this phase.

See Task 12 of Subsection 4.2.4.

13. The modeler removes any incomplete key classes and all resulting inherited attribute classes.

Either the removal of a relation class that is already in the conceptual schema (Task 4) or the change in ownership of an attribute class that is already in the conceptual schema (Tasks 7 and 9) can cause a key class member to be removed from a conceptual schema entity class, either temporarily (until Task 8 or 10) or permanently. When this happens, the key class that lost the member becomes incomplete, so it can no longer fulfill its function. Consequently, it must be removed also. The other attribute classes that are members of that key class, if any, can remain in the entity class, but their membership in that key class must be removed. Circle the key class number in the Mbr. of K.C. No. column on the Owned or Inherited Attribute Classes Form where each member appears, write "OMIT" in the margin next to it, and write "UPDATED" below the entity class number in the lower left corner.

If the key class migrated to other conceptual schema entity classes, that migration must be undone. Circle the key class number in the Ind. K.C. No. column of the Relation Classes Form for each relation class that is affected, write "OMIT" in the margin next to each, and write "UPDATED" below the entity class number in the lower left corner. If any of the affected dependent entity classes are not already in the model, add them now; see Phase 0, Task 13 for details. Write "DELETE" in the margin of the Inherited Attribute Classes Forms next to each inherited attribute class that resulted from the migration of that key class, and write "UPDATED" below the entity class number in the lower left corner.

If any of these inherited attribute classes is a key class member itself, this task must be repeated, and it must continue to be repeated until all key classes and all inherited attribute classes that are contingent on the original key class have been marked for removal. Key classes and inherited attribute classes of all affected entity classes will be re-established in Tasks 8 - 10, but they may not be exactly the same.

14. The modeler, reviewers, and librarian participate in reviewing the Phase 3 model.

See Task 5 of Subsection 4.2.4.

Deviation from IDEF1:

The specially designed Relation Classes, Owned Attribute Classes, and Inherited Attribute Classes Forms are used in place of the following regular IDEF1 forms: Related Entity Class Node Cross-Reference, Attribute Class Definition (2), and Inherited Attribute Class Cross-Reference. The forms used are designed to

facilitate loading the conceptual schema. Also, the IDEF1 forms listed below are not called for, but may be used at the discretion of the project manager:

- o Attribute Class Diagram
- o Entity Class/Attribute Class Matrix
- o Attribute Class Migration Index
- o Refinement Alternative Diagram
- o Entity Class/Function View Matrix

4.3.5 Phase 4: Defining Nonkey Attribute Classes

Objectives:

- o Identify and define the nonkey attribute classes that are within the scope of the model.
- o Identify the entity class that owns each nonkey attribute class.

This phase is patterned after Phase 4 of IDEF1, and the description presented here is less detailed than the one in the IDEF1 documentation. Please refer to that documentation for further information. Also, please refer to Subsection 4.1.3 for details on how to fill out Owned Attribute Classes Forms.

Tasks:

1. The project manager decides what method to use to review the Phase 4 model.
See Task 1 of Subsection 4.2.1.
2. The project manager schedules the Phase 4 tasks.
See Task 2 of Subsection 4.2.1.
3. The modeler populates the model with the nonkey attribute classes.
See Task 3 of Subsection 4.2.1.

Assign a tag number, prefixed with "N," to each nonkey attribute class, and record it on an Owned Attribute Classes Form. Check off each in the pool as it is used.

Also, review the name, label, and definition of each conceptual schema attribute class, record any changes that are required on the Owned Attribute Classes Form where it appears, write "CHANGE" in the margin next to it, and write "UPDATED" below the entity class number in the lower left corner.

4. The modeler defines any additional entity classes that are introduced during this phase.

Whenever a new entity class is introduced, double-check the conceptual schema to see if it is already there. Rely on the entity class definitions more than the names or labels in deciding whether a conceptual schema entity class represents the same sort of thing as a new entity class. If a new entity class is already described in the conceptual schema, prepare the modeling forms listed in Phase 0, Task 13. If it is not, immediately document it by performing the tasks in Phases 1-3 that are needed to:

- o Update the entity class pool.
 - o Prepare an Entity Class Definition Form.
 - o Update the relation class matrix.
 - o Define the relation classes that it is involved in. See Task 5 for details.
 - o Update the overview diagrams.
 - o Define and migrate its key class(es).
 - o Update any optional documents that are affected.
5. The modeler defines any additional relation classes that are introduced during this phase.

See Task 5 of Subsection 4.2.1.

6. The modeler, reviewers, and librarian participate in reviewing the Phase 4 model.

See Task 5 of Subsection 4.2.1.

Deviation from IDEF1:

The specially designed Owned Attribute Classes Form is used instead of the regular Attribute Class Definition Forms to facilitate loading the conceptual schema. Also, the following IDEF1 forms are not called for, but may be used at the discretion of the project manager:

- o Attribute Class Diagram
- o Entity Class/Attribute Class Matrix

See Subsection 5.2 for instructions on how to update the CS tables.

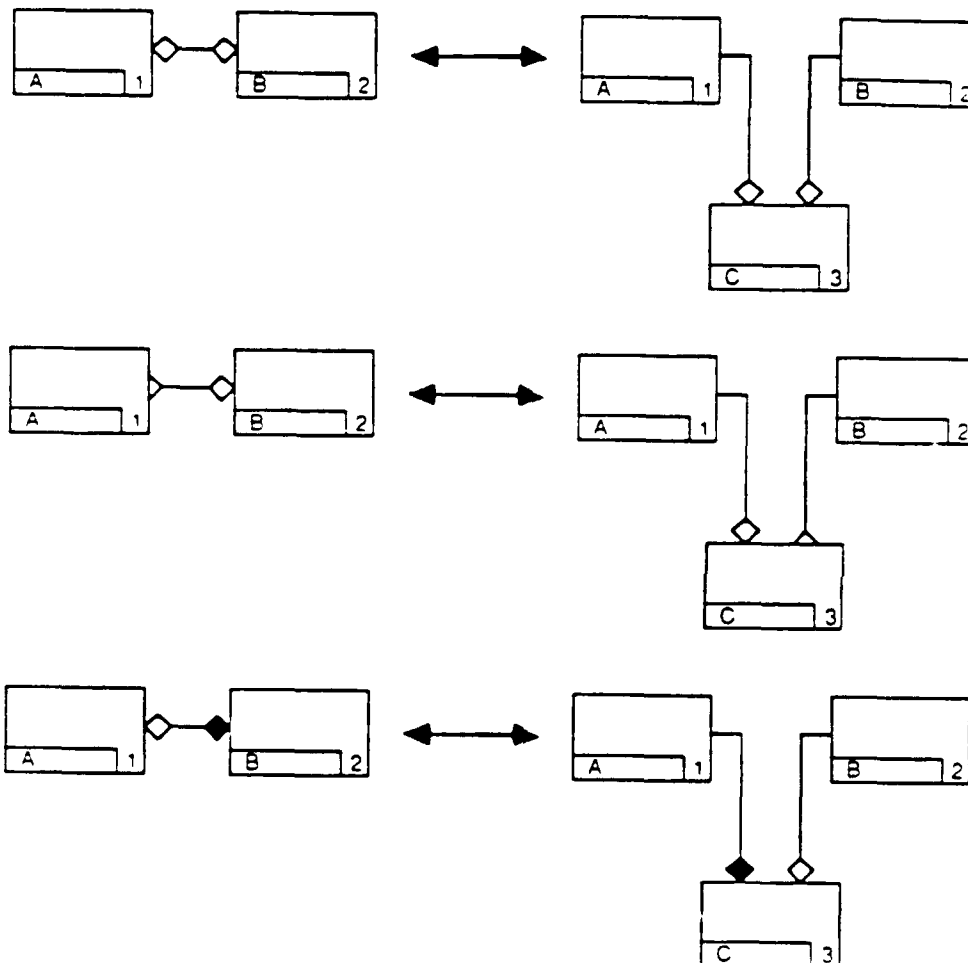


Figure 4-7. Refinements of Nonspecific Relation Classes
Example

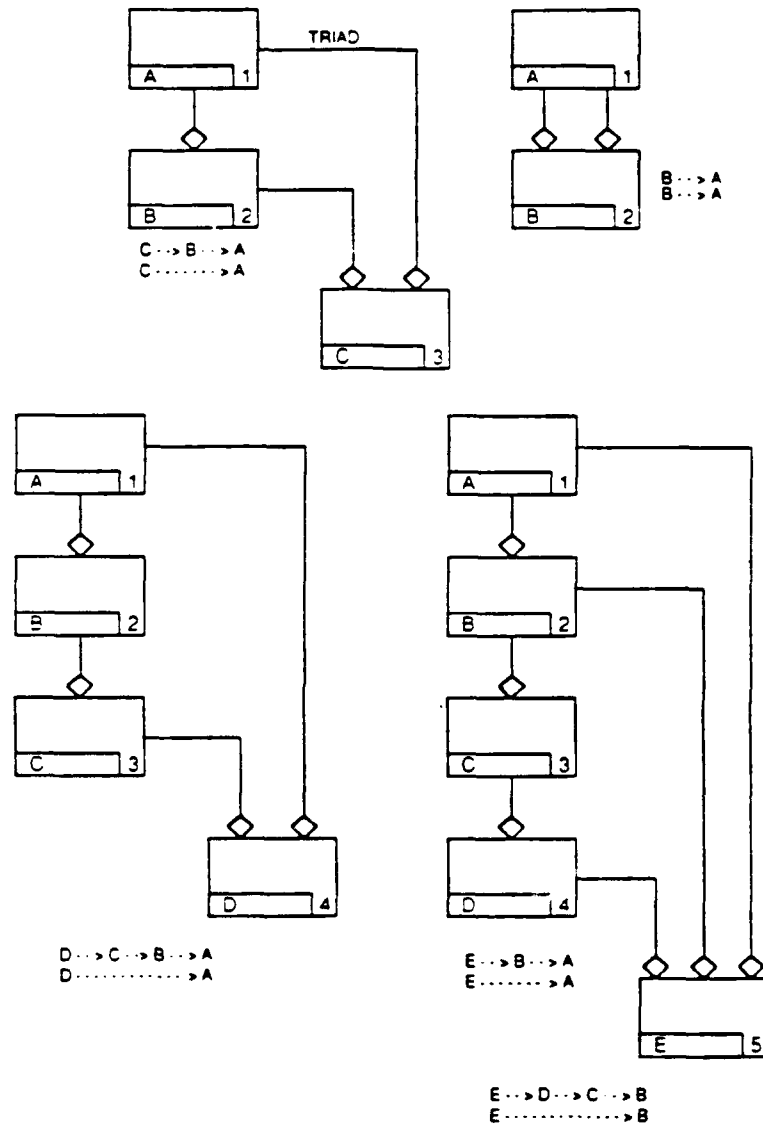


Figure 4-8. Triads and Other Dual-Path Structure Examples

Part Number is the key class of Part. It migrates through each relation class to appear twice in Component Part. The inherited attribute class that results from the left relation class could be named "Assembly Part Number; and the one from the right could be called "Component Part Number" to associate each with the appropriate relation class.

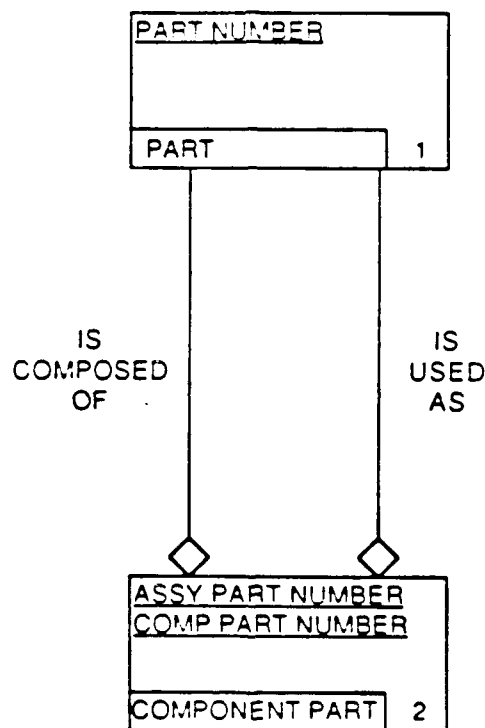


Figure 4-9. Migration Through Two Relation Classes Example

- A. In a one-to-zero-or-one relation class the key class of the dependent is usually the same as that of the independent:

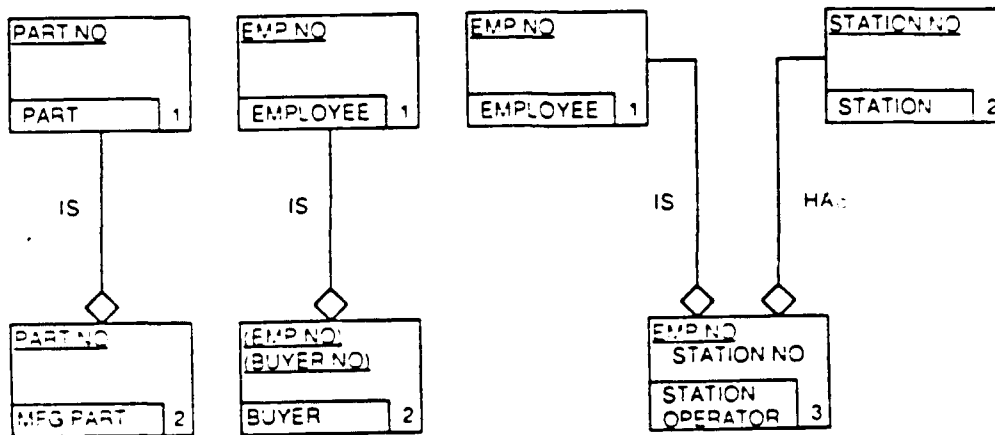


Figure 4-10. Guidelines for Determining Key Classes of Dependent Entity Classes

- B. The key class of an entity class that was derived to refine a many-to-many relation class is usually composed of attribute classes inherited from the two independent entity classes:

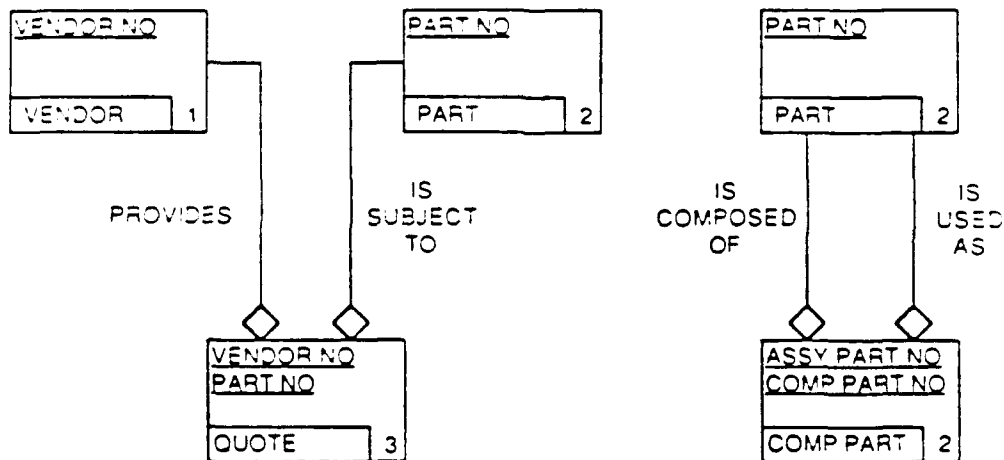


Figure 4-10. Guidelines for Determining Key Classes of Dependent Entity Classes (Continued)

- C. In this example, Bin Wrhs No and Item Wrhs No always have the same value so only one must be in the key.
- D. In this example, Proj Plan No and Tool Plant No do not always have the same value, so both must be in the key class.

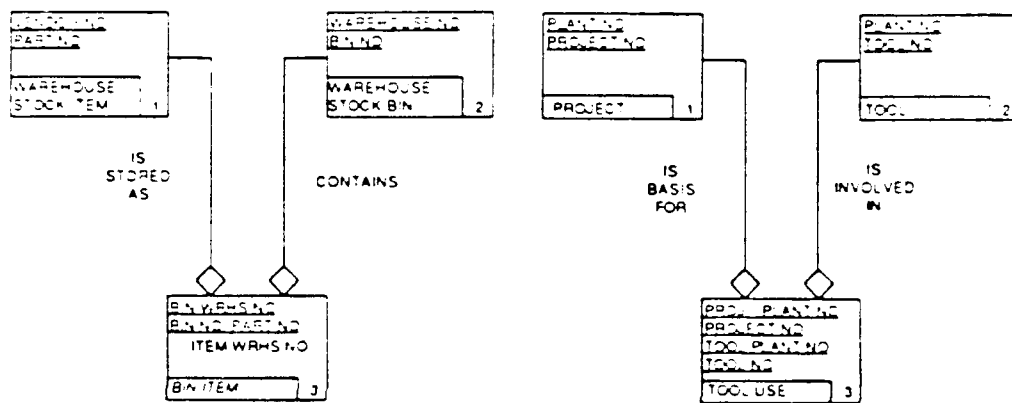


Figure 4-10. Guidelines for Determining Key Classes of Dependent Entity Classes (Continued)

SECTION 5

MAINTAINING THE CDM

5.1 Methodology Overview

This section provides the CDM Administrator with the methodology to populate and maintain the conceptual schema tables of the CDM. As was explained in Subsection 2.2.1, the CDM database is the database dictionary of the IISS. It captures knowledge of the locations, characteristics and interrelationships of all shared data in the system. The CDM database is implemented as a relational database. The information in the CDM Tables is populated and maintained using the Neutral Data Definition Language, hereafter NDDL. NDDL is an interpretive language that serves two basic purposes. It is a modeling support tool that enforces IDEF1 rules and it is a dictionary definition and maintenance tool. As detailed instructions on the use of NDDL are provided in the NDDL User's Guide. Pub. No. UM 62034110, this manual will not describe NDDL syntax and only references NDDL commands, within the context of the methodology.

The definition of the conceptual, internal and external schema objects along with their inter-schema mappings enable schema transformations to be performed. The precompiler generates software modules and code directly into the user's application to do these schema transformations.

The CDM Impact Analysis Utility identifies and reports which software modules are affected by a change to the CDM and also identifies and reports affected external schemas used by these software modules. Changes to the CDM may require:

- Modification to the application programs to work with the new CDM model
- Reprecompilation of Neutral Data Manipulation Language (NDML) software modules.
- Revision of the NDDL commands causing the CDM changes.

Whenever changes are to be made to the CDM, a CDM Impact Analysis should be run to generate reports giving information necessary as to what additional action must be taken. The CDM Impact Analysis is described in more detail in its User's Manual, Pub. No. UM 620341420.

5.1.1 Generic NDDL Commands

Figure 5-1 contains the list of CDM objects that are defined to the CDM using NDDL. They are grouped according to schema. Each object may be described with descriptive text in the CDM by using the NDDL CREATE DESCRIPTION TYPE and DESCRIBE commands. Figure 5-2 contains the CDM Tables updated by these commands. The CREATE DESCRIPTION TYPE command stores legal description types in

the DESCRIPTION_TYPE CDM Table. The DESCRIBE command refers to these description types and populates the DESC_TEXT CDM Table with descriptive text for the specified CDM object.

Also, the actual names given to these objects when they were initially defined can be changed by using the NDDL RENAME command. The CDM assigns unique identification numbers to all objects when they are defined. Changing the name of the object associated with the identifying number has no impact on any other information about that object. Only the "object"_NAME column of the appropriate CDM Table is modified.

5.1.2 Transaction NDDL Commands

Transaction NDDL commands don't populate or update the tables of the CDM database. They give the CDM Administrator control over the NDDL session. The NDDL transaction commands are:

```
SET COMMIT
COMMIT
ROLLBACK
SET OUTPUT
HALT
```

Whenever the NDDL Processor is activated to build or change the information in the CDM, the CDMA can change the commit mode from its default of "automatic". This is done by using the NDDL SET COMMIT command. After issuing this command the CDMA can gain control over the CDM database changes by issuing the NDDL COMMIT and ROLLBACK commands manually.

The NDDL COMMIT command will make the changes to the CDM database permanent. The changes are caused by any prior NDDL commands. This command is only effective if the SET COMMIT NDDL command has assigned the NDDL session to "manual" commit.

The NDDL ROLLBACK command will "un-do" any changes made to the CDM database by prior NDDL commands. This command, like the COMMIT command, is only effective if the current setting of the commit mode has been assigned to "manual". In the default "automatic" commit mode the rollback process is automatically initiated upon an error in the previous NDDL command.

The NDDL SET OUTPUT command will direct the output of certain NDDL commands to be either the screen or a file. Generated NDDL commands are the output that is directed by this command. The NDDL commands that generate NDDL as output are described further in Subsections 5.4 and 5.5.

The NDDL HALT command terminates the NDDL Session.

CONCEPTUAL SCHEMA OBJECTS

Domain
data type (Standard)
Model
Entity
Attribute
Relation

INTERNAL SCHEMA OBJECTS

Database
Host
Record
Datafield
data type
Set

EXTERNAL SCHEMA OBJECTS

View
Dataitem
data type

Figure 5-1. CDM Objects

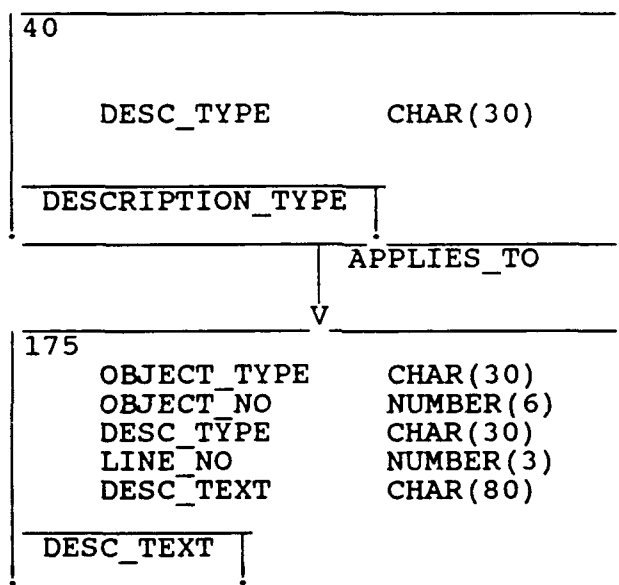


Figure 5-2. CDM Object Description

5.2 Loading the Initial CS Description

The conceptual schema's objects are defined to the CDM in the following order:

Domains
data types (Standard)
Model
Attributes
Entities
Relations

NDDL commands that define these objects to the CDM:

- a) CREATE DOMAIN
- b) CREATE ATTRIBUTE
- c) CREATE ENTITY
- d) CREATE RELATION

The conceptual schema CDM Tables that are updated by these commands are shown in Figure 5-3.

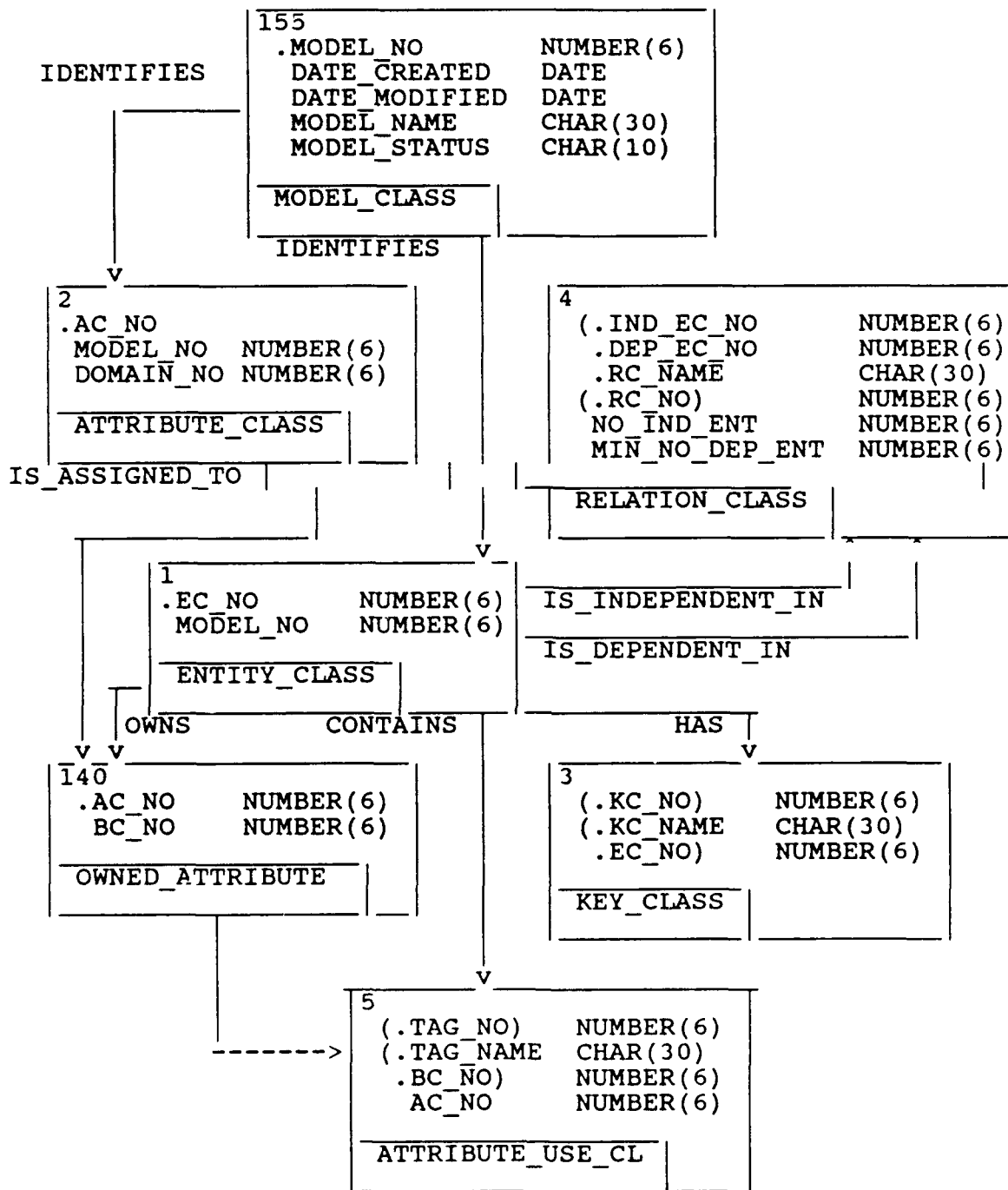


Figure 5-3. CDM Conceptual Schema

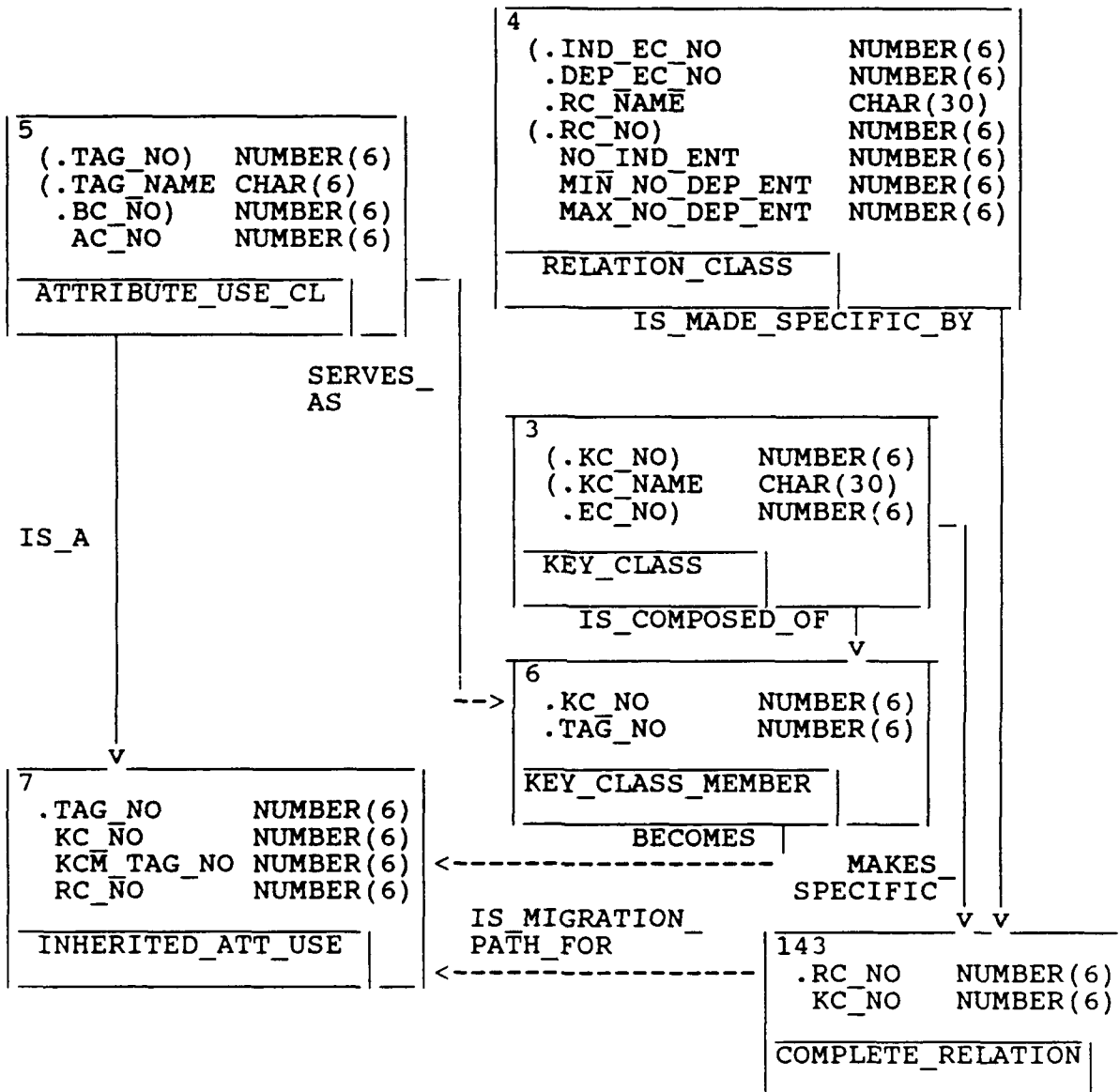


Figure 5-3. CDM Conceptual Schema (Continued)

5.2.1 Loading Domains

A Domain Class is the set of rules for setting the values and ranges allowed for an attribute. A Domain Class can have several different data types. It always has one standard data type which supplies the CDM with data storage formats for representing its attributes. Any other data types for the domain are called user defined data types which represent other formats for internal schema data fields and external schema data items. An example of a domain is "DATE". The standard data type may be defined as numeric, unsigned format, six positions long. Other formats and sizes necessary to represent the date in the enterprise's data may use Julian, a date format having a size which can be defined as a user data type.

The CDM Administrator loads the domains for the attribute classes from the Owned Attribute Classes Forms (Figure 5-4). For each attribute, use the NDDL CREATE DOMAIN command STANDARD TYPE clause to load domains and its standard data type. The allowable types of values for a standard data type are character, signed and unsigned. This and the data storage formats are indicated by the Type ID. column on the form. The CDM tables updated by the CREATE DOMAIN command are DOMAIN_CLASS and USER_DEF_DATA_TYPE.

The CDM assigns a unique number to the domain (DOMAIN_NO) and the standard data type (USDF_DT_NO). The DATA_TYPE_IND column of the USER_DEF_DATA_TYPE table is set to "STD". If the VALUE and/or RANGE clause is used in the CREATE DOMAIN command, the DOMAIN_VALUE and/or DOMAIN_RANGE CDM Table is updated and a verification module is generated. The VERIF_MODULE and SOFTWARE_MODULE CDM Tables are updated with this module's name and Id. At runtime this verification program is initiated to validate the domain values of an attribute being inserted or modified. These verification modules as well as domains can be shared across different models of conceptual schemas.

The CDM Administrator uses the NDDL DESCRIBE command with an object identifier of "DOMAIN" to load descriptions for each domain. This same command can be used with an object identifier of "data type" to load descriptions for data types.

5.2.2 Defining the Model

A model is a representation of the information requirements of all or part of an enterprise in terms of entity classes, relation classes, and attribute classes. More specifically, it is the IDEF1 model of the conceptual schema. As mentioned in Section 2, the Integration Methodology which is intended to guide the CDM Administrator in building the CDM, starts with a select portion of the enterprise's data and documents it in the conceptual schema. Subsequently other portions of the data resource are identified and modeled. The model is then loaded into the CDM database. This evolutionary approach of the conceptual schema gives rise to multiple models that are incorporated into the existing enterprise's conceptual schema, the INTEGRATED_MODEL.

To define a new model to the CDM, use the NDDL CREATE MODEL command. The MODEL CLASS CDM Table is populated with an assigned unique MODEL NO, MODEL NAME, the current date as DATE CREATED and DATE MODIFIED, and a MODEL STATUS set to "UNCHECKED". When the CDM Administrator wishes to add entity, attribute and relation classes to an existing model, the model must be established for the NDDL session with the NDDL ALTER MODEL command. Commands to merge and check models will be covered in Subsection 5.4.

5.2.3 Loading Attribute Classes

The CDM Administrator loads the attribute classes from the Owned Attribute Classes Forms (Figure 5-4) and assigns a domain to each one. Obviously, domain classes must be defined to the CDM before attribute classes can be assigned a domain. For each attribute class, use the NDDL CREATE ATTRIBUTE command and assign its domain using the DOMAIN clause. An attribute can be created without specifying a domain. The CDM assigns these attributes with a domain named "UNDEFINED". An attribute with an "UNDEFINED" domain cannot be mapped to and consequently cannot be accessed by an application. This NDDL command populates the ATTRIBUTE_CLASS and ATTRIBUTE_NAME CDM Tables.

Keywords are a way to group a number of attribute, entity or relation classes across models by using the same identifier, or keyword. For example, if attribute A, B and C model the enterprise's finance department or function, the keyword can be "FINANCE". Keywords are assigned with the KEYWORD clause of the NDDL CREATE ATTRIBUTE command. The IISS_KEYWORD and AC_KEYWORD CDM Tables are populated with this clause.

Besides its preferred name, an attribute class can have an alternate name, or more frequently referred to as a synonym in IDEF1 Methodology. The NDDL CREATE ALIAS command can assign this alternate name or alias to the attribute class. The ATTRIBUTE_NAME CDM Table is populated with this command. The CDM uses the AC_NO assigned during the CREATE ATTRIBUTE command, the alias name for the AC_NAME and sets AC_NAME_TYPE to "alias". The attribute name and its alias can then be used interchangeably in subsequent NDDL commands. Since both names represent the same attribute, the alias name of an attribute class cannot be used to define another attribute class to the CDM.

The CDM Administrator loads the descriptions for the attribute classes from the Owned Attribute Classes Forms (Figure 5-4) A.C. Definition column. For each attribute class, use the NDDL DESCRIBE command with an object identifier of "ATTRIBUTE".

| | | | | | | | | | | |
|---|----------------------------|------------|------|-----------|---|--------------|-----------------|------|---------|--|
| USED AT | AUTHOR | A C Nowlin | DATE | 23 Dec 82 | X | WORKING | REVIEW | DATE | CONTEXT | |
| | PROJECT | | REV | | | DRAFT | | | | |
| | NOTES 1 2 3 4 5 6 7 8 9 10 | | | | | RECOMMENDED | | | | |
| | | | | | | FINALIZATION | | | | |
| <p>Entity Class Name User Assignment</p> <p>Entity Class Label User Assign</p> <p>Entity Class Definition Specific individuals are assigned responsibilities for a variety of roles Such assignment allows decision to be made in a controllable manner Each assigned individual is uniquely identified by a user identifier</p> | | | | | | | | | | |
| MODE | E35 G1 | TITLE | | | | | User Assignment | | NUMBER | |

Figure 5-4. Owned Attribute Classes Form Example

5.2.4 Loading Entity Classes

The CDM Administrator loads entity classes into the CDM using the Entity Class Glossary Form (Figure 5-5) and the Owned Attribute Classes Form (Figure 5-4) which have the same "NODE" (lower left corner of form). For each Entity Class Glossary Form define the entity class to the CDM using the NDDL CREATE ENTITY command. The CDM assigns a unique EC_NO to the entity being defined and populates the ENTITY_CLASS and ENTITY_NAME CDM Table. Assign owned attributes to the entity class by using the OWNED ATTRIBUTE clause of the command. An attribute can be owned by only one entity in the model. Obviously the attributes assigned as owned to the entity must have been defined to the CDM, prior to this step. The OWNED_ATTRIBUTE and ATTRIBUTE_USE_CL CDM Tables are populated by this clause.

Keywords are a way to group a number of attribute, entity or relation classes across models by using the same identifier, or keyword. For example, if entity ENTA, ENTB and ENTC model the enterprise's Payroll department or function, the keyword can be "PAYROLL". Keywords are assigned with the KEYWORD clause of the NDDL CREATE ENTITY command. The IISS_KEYWORD and EC_KEYWORD CDM Tables are populated with this clause.

Besides its preferred name, an entity class can have an alternate name, or more frequently referred to as a synonym in IDEF1 Methodology. The NDDL CREATE ALIAS command can assign this alternate name or alias to the entity class. The ENTITY_NAME CDM Table is populated with this command. The CDM uses the EC_NO assigned during the CREATE ENTITY command, the alias name for the EC_NAME and sets EC_NAME_TYPE to "ALIAS". The entity name and its alias can then be used interchangeably in subsequent NDDL commands. Since both names represent the same entity, the alias name of an entity class cannot be used to define another entity class to the CDM.

The CDM Administrator loads the descriptions for the entity classes from the Entity Class Glossary Form's (Figure 5-5) Entity Class Definition area. For each entity class, use the NDDL DESCRIBE command with an object identifier of "ENTITY".

Another optional clause of the CREATE ENTITY command is KEY CLASS which defines a key class for the entity. This option is provided for a model that is not developed in the recommended IDEF1 phases, but is explained in Subsection 5.2.5.

| | | | | | | |
|----------------------------|-------------------|-------------|--------------|---------|------|---------|
| USED AT | AUTHOR PROJECT | DATE REV | WORKING | RELEASE | DATE | CONTEXT |
| | | | DRAFT | | | |
| | | | RECOMMENDED | | | |
| | | | FINALIZATION | | | |
| NOTES 1 2 3 4 5 6 7 8 9 10 | | | | | | |

| Tag No | Tag & Label | A C No | Ind. E C No | Ind. K C No | Ind. Tag No | Migration Path R C Label | Mbr of K C No |
|--------|--|--------|-------------|-------------|-------------|--------------------------|---------------|
| T105 | Operation Plan Identification (OP Plan ID) | A68 | E11 | K1 | T34 | Initiates | |
| T106 | Issuing Resource Identification (Iss Resource ID) | A76 | E21 | K1 | T59 | Issues | |
| T107 | Benefiting Resource Identification (Ben Resource ID) | A76 | E21 | K1 | T59 | Will Benefit From | |
| T108 | Stock Area Identification (Stock Area ID) | A83 | E30 | K1 | T42 | Is Depleted By | |
| T109 | Item Identification (Item ID) | A24 | E30 | K1 | T43 | Is Depleted By | |

| | | | | | |
|------|-----|-------|-----------------------------|--------|--|
| MODE | E20 | TITLE | Inherited Attribute Classes | NUMBER | |
|------|-----|-------|-----------------------------|--------|--|

Figure 5-5. Entity Class Glossary Form Example

5.2.5 Loading Key Classes and Relation Classes

The CDM Administrator loads the key and relation classes using the final IDEF1 Model Diagram, Inherited Attribute Use Class Form (Figure 5-6) and the Relation Class Form (Figure 5-7). For each entity class of the final IDEF1 model; in top-down order (level by level) for each entity at that level:

- Use the NDDL ALTER ENTITY command and the optional ADD KEY clause to name the key class and specify the attributes that comprise the entity's key. The KEY_CLASS and KEY_CLASS_MEMBER CDM Tables are populated by the ADD KEY clause.

- Define to the CDM each relation class originating from this entity by using the NDDL CREATE RELATION command. The RELATION_CLASS CDM Tables is populated by this command. Migrate the key class which was just defined by the prior step, using the MIGRATES clause. It is recommended that the CDM Administrator designates one key class as the primary key and migrates this key to all its dependent entities in order to provide for an easy transition to the IDEF1X methodology. Migrating the key class completes the relation and populates the COMPLETE_RELATION, ATTRIBUTE_USE_CL and INHERITED_ATT_USE CDM Tables. The MIGRATES clause of the CREATE RELATION command also allows the option of renaming the tag names (i.e. providing role names) of the key class attributes in the dependent entity. Use of role names become necessary when defining relations in a Bill of Materials Structure.

Keywords are a way to group a number of attribute, entity or relation classes across models by using the same identifier, or keyword. For example, if relation classes DEPARTMENT ENT SERVES AS PROD_CENTER_ENT and MACHINE_ENT MAKES_SPECIFIC_TOOLS_ENT model the enterprise's Manufacturing department or function, the keyword can be "MANUFACTURING". Keywords are assigned with the KEYWORD clause of the NDDL CREATE RELATION command. The IISS_KEYWORD and RC_KEYWORD CDM Tables are populated with this clause.

The CDM Administrator loads the descriptions for the relation classes by using the NDDL DESCRIBE command with an object identifier of "RELATION".

| | | | | | | | | | | | | | |
|---------|--|----------------------------|--|----------|--|--------------|--|--------|--|------|--|---------|--|
| USED AT | | AUTHOR PROJECT | | DATE REV | | WORKING | | READER | | DATE | | CONTEXT | |
| | | | | | | DRAFT | | | | | | | |
| | | NOTES 1 2 3 4 5 6 7 8 9 10 | | | | RECOMMENDED | | | | | | | |
| | | | | | | FINALIZATION | | | | | | | |

| Tag No | A.C. Name & Label | A.C. No | A.C. Definition | Type ID | Mbr of K.C. No |
|--------|----------------------------------|---------|--|---------|----------------|
| T17 | Location Identification (Loc ID) | A08 | Unique Identification Assigned To Each Location Where Items Are Stored | C(8) | K01 |

| | | | | | |
|------|----|-------|-------------------------|--------|--|
| MODE | EB | TITLE | Owned Attribute Classes | NUMBER | |
|------|----|-------|-------------------------|--------|--|

Figure 5-6. Inherited Attribute Classes Form Example

| | | | | | | |
|----------------------------|---------|---------------|-------------|----------|------|---------|
| USED AT | AUTHOR | DATE: REV. | WORKING | IF ADDED | DATE | CONTEXT |
| | PROJECT | | DRAFT | | | |
| | | | RECOMMENDED | | | |
| | | | PUBLICATION | | | |
| NOTES 1 2 3 4 5 6 7 8 9 10 | | | | | | |

| Tag No. | Tag & Label | A.C. No. | Ind. E.C. No. | Ind. K.C. No. | Ind. Tag No. | Migration Path R.C. Label | Mbr. of K.C. No. |
|---------|---|----------|---------------|---------------|--------------|---------------------------|------------------|
| T16 | Storage Area Identification (Star Area ID) | A07 | E29 | K01 | T13 | Is Composed Of | K01 |
| T194 | Storage Location Status (Loc Status) | A18 | E72 | K01 | T178 | Is Assigned To | |

| | | | | | |
|------|----|-------|-----------------------------|--------|--|
| MODE | E8 | TITLE | Inherited Attribute Classes | NUMBER | |
|------|----|-------|-----------------------------|--------|--|

Figure 5-7. Relation Classes Form Example

5.3 Modifying/Deleting CS Objects

Prior to modifying or deleting elements of the conceptual schema, the CDM Administrator must assess the impact of the proposed change on the other components of the CDM. As stated earlier, whenever changes are to be made to the CDM, a CDM Impact Analysis should be run to generate reports giving information necessary to determine what additional action must be taken. Refer to the CDM Impact Analysis Use Manual for instructions on how to use the Impact Analysis Tool.

The objective of this subsection is to provide the CDM Administrator with the information necessary to make these changes, determine the prerequisites before the changing or dropping CS objects, and being aware of the options available when changing each CS Object. As a general rule, the NDDL processor does not allow CS objects to be dropped if mappings exist to the internal or external schemas.

The following NDDL commands are used for modifying and deleting the conceptual schema objects (i.e., domain, standard data type, model, attribute, entity and relation classes):

```
ALTER DOMAIN
DROP DOMAIN
ALTER MODEL
DROP MODEL
ALTER ATTRIBUTE
DROP ATTRIBUTE
ALTER ENTITY
DROP ENTITY
ALTER RELATION
DROP RELATION
```

5.3.1 Domain Class Changes

The values and ranges specified in a domain class govern the content of attribute classes in schema transformations. The standard data type stipulates the data storage formats for the conceptual schema object - attribute class. Domain classes are modified using the NDDL ALTER DOMAIN command. The following domain changes apply to the conceptual schema:

- * Change a user defined data type to the standard data type.

Use the ALTER DATA TYPE clause. The DATA_TYPE_IND in the USER_DEF_DATA_TYPE CDM Table is changed from "user" to "std". This clause also permits the representation format to be altered.

- * Change the valid values and ranges for the attribute class associated with the standard data type of a domain class.

Use the ADD VALUE, ADD RANGE, DROP VALUE, and/or DROP RANGE clauses to add and delete valid values and ranges. The DOMAIN_VALUE and DOMAIN_RANGE CDM Tables are updated with this clause. Another verification module is

generated and the VERIF_MODULE and SOFTWARE_MODULE CDM Tables are updated with the generated software module's name and Id.

Domain Classes are deleted from the CDM by the NDDL DROP DOMAIN command. Before a domain class can be dropped:

- * Change the domain assignment of any attributes assigned to the domain being deleted.

Use the NDDL ALTER ATTRIBUTE command DOMAIN clause.

-OR-

- * Drop all attributes that use this domain.

Use the NDDL DROP ATTRIBUTE command.

- * Change the data type assignment of any internal schema datafields using any data types of the domain to be dropped.

Use the NDDL ALTER FIELD command.

-OR-

- * Drop all internal schema datafields that use any data types of the domain to be dropped.

Use the NDDL DROP FIELD command.

- * Change the data type assignment of any external schema data item using any data types of the domain to be dropped.

Use the NDDL DROP VIEW command, then re-create the view assigning a different data type to the data item using the CREATE VIEW command.

- * Drop all external schema data items that use any data types of the domain to be dropped.

Use the NDDL DROP VIEW command.

- * Change the data type assignment for all module parameters specifying any data types of the domain to be dropped.

Use the NDDL ALTER MODULE command.

-OR-

- * Drop all module parameters that use any data types of the domain to be dropped.

Use the NDDL DROP MODULE COMMAND.

The DROP DOMAIN command will drop all data types associated with the domain from the USER_DEF_DATA_TYPE CDM Table. Any values and/or ranges associated with the domain class will be

deleted from the DOMAIN_VALUE and DOMAIN_RANGE CDM Tables. The generated verification module entry will be dropped from the VERIF_MODULE and the SOFTWARE_MODULE CDM Tables.

5.3.2 Model Changes/Deletes

Models are modified by adding and/or dropping entity, attribute and relation classes to an existing model. Before a model can be modified:

- * The model must be established as current for the NDDL session.

Use the NDDL ALTER MODEL command. All subsequent NDDL conceptual schema commands will be applied to the current model.

All models, except the INTEGRATED MODEL can be dropped from the CDM. This is performed with the NDDL DROP MODEL command. The CDM Impact Analysis will report all software modules and external views affected by the deleted model. Before a model can be deleted:

- * Drop all software modules reported that use the external views comprised of attributes and entities of the model.

Use the NDDL DROP MODULE command.

- * Drop all complex mapping algorithms of the model.

Use the NDDL DROP ALGORITHM command.

- * Drop all external views used in these modules.

Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for attribute classes of the model.

- * Drop all the CS-IS mappings including partitions and unions that map to the cs objects of the model.

Use the NDDL DROP MAP, DROP PARTITION and DROP UNION commands.

When the model is deleted with the NDDL DROP MODEL command, everything associated with the model will be dropped from the following CDM Tables:

MODEL_CLASS
ATTRIBUTE_CLASS
ATTRIBUTE_NAME
ENTITY_CLASS
ENTITY_NAME
RELATION_CLASS
OWNED_ATTRIBUTE
ATTRIBUTE_USE_CLASS
INHERITED_ATT_USE

KEY_CLASS
KEY_CLASS_MEMBER
COMPLETE_RELATION

Also, all descriptions, aliases and keywords for the entities, attributes and relations of the model will be dropped from the CDM. Entries for the model in the AC_KEYWORD, EC_KEYWORD RC_KEYWORD and DESC_TEXT CDM Tables are deleted.

5.3.3 Attribute Class Changes/Deletes

The CDM Administrator can modify an attribute class with the NDDL ALTER ATTRIBUTE command. The changes that can be made with this command are:

- * Change the attribute class domain assignment

Use the DOMAIN clause. The CDM Table ATTRIBUTE_CLASS will be updated with the new DOMAIN_NO.

- * Change or delete the attribute class keyword

Use the ADD KEYWORD and/or DROP KEYWORD clauses to add and/or delete keyword references for the attribute class. The AC_KEYWORD CDM Table is updated.

- * Change the entity class that owns the attribute class

Use the OWNERSHIP to ENTITY clause. The CDM Impact Analysis will report all software modules and external views affected by this NDDL command. Before ownership of the attribute can be altered:

- Drop all software modules reported that use the external views comprised of this attribute class. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use the attribute class as an input or an output parameter. Use the NDDL DROP ALGORITHM command.
- Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the attribute class.
- Drop all the CS-IS mappings including partitions and unions that map to the attribute class. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION commands.

Attribute classes are deleted from the CDM by the NDDL DROP ATTRIBUTE command. The CDM Impact Analysis will report all software modules and external views affected by this NDDL command. Before the attribute class can be dropped:

- Drop all software modules reported that use the external views comprised of this attribute class. use the NDDL DROP MODULE command.

- Drop all complex mapping algorithms that use this attribute class as an input or output parameter. Use the NDDL DROP ALGORITHM command.
- Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the attribute class.
- Drop all the CS-IS mappings including partitions and unions that map to the attribute class. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION command.

The DROP ATTRIBUTE command will drop all entries for the attribute class in the ATTRIBUTE_CLASS, OWNED_ATTRIBUTE, ATTRIBUTE_USE_CLASS, KEY_CLASS_MEMBER, ATTRIBUTE_NAME, AC_KEYWORD and INHERITED_ATT_USE CDM Tables. If the attribute class was the last member in a key class, the entry is deleted in the KEY_CLASS CDM Table. All migrations of this attribute are also deleted. Refer to Subsections 5.3.4 and 5.3.5 for more information on modifying an attribute that is owned by an entity, participates in a key class or is migrated through a relation class.

5.3.4 Entity Class Changes/Deletes

This section covers modifications made to an entity class, its owned attributes and its key class. The CDM Administrator can modify an entity class with the NDDL ALTER ENTITY command. The changes that can be made with this command are:

- * Establish a new key class for the entity

Use the ADD KEY clause. The CDM Tables KEY_CLASS and KEY_CLASS_MEMBER will be updated.

- * Assign attributes as owned by the entity

Use the ADD OWNED ATTRIBUTE clause. If the attribute class being assigned as owned has not been created with a CREATE ATTRIBUTE command, a new ATTRIBUTE_CLASS entry is created as well as an entry in the ATTRIBUTE_USE_CL and OWNED_ATTRIBUTE CDM tables for the entity class. An attribute can be owned by only one entity in a model.

- * Delete a key class from the entity

Use the DROP KEY clause. The entry for the key class is deleted from the KEY_CLASS CDM Table. All KEY_CLASS_MEMBER entries for the key class are deleted. All ATTRIBUTE_USE_CL, INHERITED_ATT_USE and COMPLETE_RELATION entries that were created by the migration of this key class are deleted. The CDM Impact Analysis Utility will report all software modules and external views affected by this NDDL command. Before the key class of the entity can be dropped:

- Drop all software modules reported that use the external views comprised of the attributes belonging to this key class. Use the NDDL DROP MODULE command.

- Drop all complex mapping algorithms that use the attributes belonging to this key class as input or output parameters. Use the NDDL DROP ALGORITHM command.
 - Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the attributes that are key class members.
 - Drop all the CS-IS mappings including partitions and unions, that map to the attributes that are key class members. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION command.
- * Drop attribute classes that are owned by the entity class
- Use the DROP OWNED ATTRIBUTE clause. Entries in the OWNED_ATTRIBUTE and ATTRIBUTE_USE_CL CDM Tables are deleted for the attribute being dropped from the entity class. All migrations of this attribute, if any, will also be deleted. The CDM Impact Analysis will report all software modules and external views affected by this NDDL command. Before the owned attributes of this entity class can be dropped:
- Drop all software modules reported that use the external views comprised of the owned attribute classes being dropped. Use the NDDL DROP MODULE command.
 - Drop any complex mapping algorithms that use the owned attribute class as an input or output parameter. Use the NDDL DROP ALGORITHM command.
 - Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the attributes classes being dropped.
 - Drop all the CS-IS mappings including partitions and unions that map to the attributes that are being dropped. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION command.
- * Change or delete the entity class keyword
- Use the ADD KEYWORD and/or DROP KEYWORD clauses to add and/or delete keyword references for the entity class. The EC_KEYWORD CDM Table is updated.
- * Change the tag names of attribute classes that belong to the entity class being altered.
- Use the ALTER ATTRIBUTE clause. The TAG_NAME or role name of the ATTRIBUTE_USE_CL CDM Table is changed to the new name.
- * Change a key class of the entity class being altered.

Use the ALTER KEY CLASS clause. This clause allows the key class name to be changed. Also additions, deletions and substitutions can be made to the key class members.

The KEY_CLASS and KEY_CLASS_MEMBER CDM Tables are updated. Entries in the INHERITED ATT USE CDM Tables caused by the key class migration are deleted if key class members are dropped. The key class member (attribute class) still belongs to the entity class, though it isn't part of the entity's key class any longer.

Entity classes are deleted from the CDM by the NDDL DROP ENTITY command. The CDM Impact Analysis will report all software modules and external views affected by this NDDL command. Before the entity class can be dropped:

- Drop all software modules reported that use the external views comprised of this entity class. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use the attribute classes of the entity class to be dropped as input or output parameters. Use the NDDL DROP ALGORITHM command.
- Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the entity class.
- Drop all the CS-IS mappings including partitions and unions that map to the attributes that belong to the entity class. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION command.

The DROP ENTITY command will drop all entries for the entity class in the ENTITY_CLASS, OWNED ATTRIBUTE, ATTRIBUTE USE CLASS, INHERITED ATT USE, KEY_CLASS, KEY_CLASS_MEMBER, ENTITY_NAME, and EC_KEYWORD CDM Tables. If the entity class was involved in a relation class, the entry is deleted in the RELATION_CLASS CDM Table. Please note that dropping an entity will cause its dependent entity chain to be deleted, along with these entity's attributes, keys and relations.

5.3.5 Relation Class Changes/Deletes

The CDM Administrator can modify a relation class with the NDDL ALTER RELATION command. The changes that can be made with this command are:

- * Change the cardinality of the relation class.

If any of the cardinality integers specified in the ALTER RELATION command are different than the original cardinality integers specified when the relation was created, the cardinality of the relation class is modified. The RELATION_CLASS CDM Table is updated with the new cardinality.

* Migrate a key class from the independent entity class.

Use the ADD MIGRATES clause. The ATTRIBUTE USE CL and INHERITED ATT USE CDM Tables are added to for each key class member that migrated to the dependent entity. The COMPLETE RELATION CDM Table is updated. This clause also allows the tag names of the key class members to be changed in the dependent entity when the key class is migrated.

* Delete key class migrations through a relation class

Use the DROP MIGRATES clause. Entries in the ATTRIBUTE USE CL and INHERITED ATT USE CDM Tables for key class members previously migrated to their dependent entity are deleted. The CDM Impact Analysis Utility will report all software modules and external views affected by this NDDL command. Before the key class migration can be dropped:

- Drop all software modules reported that use the external views comprised of the inherited attribute classes being deleted because the key class migration is being dropped. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use these dropped inherited attribute classes as input or output parameters. Use the NDDL DROP ALGORITHM command.
- Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the inherited attribute classes being deleted.
- Drop all the CS-IS mappings including partitions and unions that map to the inherited attributes being deleted by dropping the key class migration. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION command.

* Change or delete the relation class keyword.

Use the ADD KEYWORD and/or DROP KEYWORD clauses to add and/or delete keyword references for the relation class. The RC_KEYWORD CDM Table is updated.

Relation classes are deleted from the CDM by the NDDL DROP RELATION command. The DROP RELATION command will drop the entries for the relation class in the RELATION CLASS and COMPLETE RELATION CDM Table. The key class that was migrated through the relation class is "unmigrated". The entries or the dependent entity in the ATTRIBUTE USE CLASS and INHERITED ATT USE CDM Tables are deleted. Please note that deleting a relation can cause deletion of all inherited attributes down the migration chain. The CDM Impact Analysis will report all software modules and external views affected by this NDDL command. Before the relation class can be dropped:

- Drop all software modules reported that use the external views comprised of the inherited attributes of the "unmigrated" keys. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use the inherited attributes of the "unmigrated" keys as input or output parameters. Use the NDDL DROP ALGORITHM command.
- Drop all external views used in these modules. Use the NDDL DROP VIEW command. This command deletes CS to ES mappings that exist for the inherited attributes that are not deleted because of the "unmigrated" keys.
- Drop all the CS-IS mappings including partitions and unions that map to the inherited attributes that will be deleted when the key classes are "unmigrated". Use the NDDL DROP MAP, PARTITION and UNION command.
- Drop all the relation class to set mappings that exist for the relation class. Use the NDDL DROP MAP command.

5.4 Modeling & Validating Tools

Several NDDL commands have been designed and developed as modeling tools and to help the CDM administrator validate models. The output of these commands can be directed to a file or the screen (see SET OUTPUT in Section 5.1.2) then reviewed, edited and resubmitted. These NDDL commands are:

CHECK MODEL
COMBINE ENTITY
COMPARE MODEL
MERGE MODEL

Refer to the NDDL User's Manual for the syntax and semantics of these commands. Section V of the NDDL User's Manual displays examples for each of these NDDL commands.

Another modeling tool is the CDM Compare Utility. It should be used by the CDM Administrator, after making changes to the CDM, to ensure the CDM remains in a consistent state. The CDM Compare Utility is used to compare two versions of a CDM and report their differences. Refer to the CDM Compare Utility User's Manual for information on its use.

5.5 Reviewing the Contents of the CDM

Specific NDDL commands were developed to copy all the information recorded in the CDM. These are the NDDL COPY commands. If there is information recorded in the CDM, there is an NDDL COPY statement that will allow that information to be copied. The copied information (or output of these commands) is in the form of NDDL statements that, if re-submitted to the NDDL processor, will again produce the information stored in the CDM. This output can be directed either to a screen or a file. Other optional clauses allow related objects to be copied along with the CDM object initially specified on the command. Refer to the NDDL User's Manual for the specific syntax and optional clauses of the command.

Another method of reviewing the information stored in the CDM is to initiate the CDM Reports. These display the CDM contents of the specified conceptual, internal or external schema definition in report format. Refer the CDM Reports User's Manual or the formats and commands to initiate these reports.

SECTION 6

MAINTAINING INTERNAL SCHEMAS AND MAPPINGS

6.1 Methodology Overview

This section provides the CDM Administrator with the methodology for building and maintaining internal schemas and for mapping them to the conceptual schema. The tables of the CDM database that are populated to describe internal schemas and CS-IS mappings are illustrated in the applicable subsections. As mentioned in Section 5, the internal schema objects are:

- Database
- Host
- Record
- Datafield
- data type
- Set

Each object represents a CDM database table which is populated using NDDL. There are various generic database models (CODASYL, relational, hierarchical, etc.) and most database management systems (DBMSs) are based on one or another of them though the terminology may be different. These objects are common to all DBMS' models. The mapping between the conceptual schema and internal schema has three levels:

- 1) Entity class to record type
- 2) Relation class to record set relationship
- 3) Attribute use class to data field

This section will explain how to define existing physical database models to the CDM and how to determine their mappings to the conceptual schema. The database models and their DBMSs whose definitions are supported by the CDM are:

- a) Relational (ORACLE, DB2)
- b) CODASYL (VAX-11, IDMS, IDS-11)
- c) Network (TOTAL)
- d) Hierarchical (IMS)

Of these DBMSs that control the various database models, only ORACLE, DB2 and VAX-11 have been fully implemented by the IISS precompiler. Therefore, this section will explain the definition of database models created and maintained by these DBMSs in-depth. Differences in network and hierarchical database models will be addressed in Section 6.5, "Specific Considerations". Forms are used to assist in the CS-IS mappings and instructions on their use are provided. The NDDL commands necessary to initially load internal schemas and CS-IS mappings are described along with the NDDL commands to change the internal schema objects.

This methodology does not address the creation of physical database designs. DBMS vendors, books, classes, etc., offer much more guidance in this area than can be provided here.

6.1.1 Internal Schema and CS-IS Mapping Structure

As mentioned in the previous subsection, the various generic database models are the basis for most DBMSs. In addition, a particular model may be modified or extended for a particular DBMS. Each of these models generates its own style of internal schema. While many internal schema components are common to all styles, some are peculiar to only one or a few. The CDM does not contain a separate structure for each style of internal schema. Instead, a single, composite structure that can support any style is provided. Each internal schema component is conceptually represented by one entity class regardless of how many styles that component is in. Additional relevant entity classes for each style are listed in the appropriate subsection of Section 6.5, "Specific Considerations".

A CS-IS mapping is intended to show which components of an internal schema correspond to those of the conceptual schema. A record type maps to an entity class if they both represent the same kind of "real-world" things. For example, in Figure 6-1, the EMP-MAST record type maps to the Employee entity class because both represent employees. There is a one-for-one correspondence between the record type and the entity class; each employee is represented by one instance of the record type and by one instance of the entity class. Notice that even though the record type contains data fields (DIV-NO, DEPT-NAME, SPOUSE-NAME) that correspond to attribute use classes in other entity classes, the record type does not map to those other entity classes. It represents a different kind of "real-world" things than any of those entity classes and is not in a one-for-one correspondence with any of them. For example, one instance of the Department entity class exists for each department while several instances of the EMP-MAST record type exist, one for each employee in a department, or if a department has no employees, no record type instances exist for that department. As another example, the Married Employee entity class has an instance for each employee who is married, while the EMP-MAST record type has an instance for every employee, married or not.

CONCEPTUAL SCHEMA

INTERNAL SCHEMA

| | |
|---------------|--|
| <u>DIV NO</u> | |
| DIV NAME | |
| DIVISION | |

HAS
< >

| | |
|----------------|--|
| <u>DEPT NO</u> | |
| DEPT NAME | |
| DIV NO | |
| DEPT | |

HAS
< >

| | |
|---------------|--|
| <u>EMP NO</u> | |
| EMP NAME | |
| DEPT NO | |
| EMPLOYEE | |

IS
< >

| | |
|---------------|--|
| <u>EMP NO</u> | |
| SPOUSE NAME | |
| MARRIED | |
| EMPLOYEE | |

EMP_MAST

| | |
|---------------|--|
| <u>EMP NO</u> | |
| EMP_NAME | |
| DIV_NO | |
| DEPT_NO | |
| DEPT_NAME | |
| SPOUSE_NAME | |

MAPS TO
<----->

Figure 6-1. Entity Class/Record Type Mapping

In a similar manner, a data field maps to an attribute use class if they both represent the same kind of data about "real-world" things. Using the example in Figure 6-1 again, the EMP-NO, EMP-NAME, and DEPT-NO data fields in the EMP-MAST record type map to attribute use classes in the Employee entity class; DIV-NO and DEPT-NAME, to those in the Dept entity class; and SPOUSE-NAME, to one in the Married Employee entity class. Notice that some data fields could map to more than one attribute use class. For example, EMP-NO and DEPT-NO could have mapped to attribute use classes in the Married Employee and Department entity classes, respectively, instead of those in the Employee entity class. They map to those in the Employee entity class because the record type maps to that entity class. DIV-NO is another example; it could have mapped to an attribute use class in the Division entity class rather than to one in the Department entity class. The reason it maps to the one in the latter is that the Department entity class is more closely related to the Employee entity class than the Division entity class is. Notice also that all these examples involve attribute use classes that belong to key classes. This is because only they can migrate to other entity classes; an owned, nonkey attribute use class appears only in its owner entity class. These situations are summarized in the following mapping rules:

- o If a data field could map to either an attribute use class in the entity class to which the record type maps or to one in another entity class, it always maps to the former (e.g., EMP-NO and DEPT-NO).
- o If a data field could map to more than one attribute use class, none of which are in the entity class to which the record type maps, it always maps to the one in the entity class that is most closely related to the entity class to which the record type maps (e.g., DIV-NO).

Finally, a record set maps to a relation class if they both represent the same kind of association between "real-world" things. This implies that the two record types in the record set, the owner and the member, map to the two entity classes in the relation class, the independent and the dependent, respectively.

The following subsections (6.1.1.1 - 6.1.1.6) present various subjects to consider when dealing with CS-IS mappings.

None of them are mutually exclusive; each can be combined with one or more of the others.

6.1.1.1 Vertical Partitions

An entity class is vertically partitioned when some of its attribute use classes map to data fields in one record type and others map to those in another. An entity class can have several vertical partitions. Each record type maps to the entity class.

CONCEPTUAL SCHEMA

| |
|-----------|
| EMP_NO |
| EMP_NAME |
| EMP_SKILL |
| EMP |

INTERNAL SCHEMA

| |
|----------|
| EMP_NO |
| EMP_NAME |

| |
|-----------|
| EMP_NO |
| EMP_SKILL |

6.1.1.2 Horizontal Partition

An entity class is horizontally partitioned when some of its entity instances map to instances of one record type and others map to instances of another record type. Usually, the horizontal partitioning of an entity class is governed by the values in a particular attribute use class.

The attribute use class governing the partition may be in the entity class being partitioned (as in the first example below), or it may be in one that that entity class is dependent on, either directly or indirectly (as in the subsequent example below).

CONCEPTUAL SCHEMA

INTERNAL SCHEMA

CALIF. EMPLOYEE

ARIZ. EMPLOYEE

| | |
|----------|--|
| EMP NO | |
| EMP NAME | |
| STATE | |
| EMPLOYEE | |

MAPS TO
<----->

| | |
|----------|--|
| EMP NO | |
| EMP_NAME | |

| | |
|----------|--|
| EMP NO | |
| EMP_NAME | |

| | |
|------------|--|
| DIV NO | |
| DIV NAME | |
| STATE_NAME | |
| DIVISION | |

< >

| | |
|---------|--|
| DEPT NO | |
| DIV NO | |
| DEPT | |

< >

CALIF. EMPLOYEE

ARIZ. EMPLOYEE

| | |
|----------|--|
| EMP NO | |
| EMP NAME | |
| DEPT NO | |
| EMPLOYEE | |

MAPS TO
<----->

| | |
|----------|--|
| EMP NO | |
| EMP_NAME | |
| DEPT_NO | |

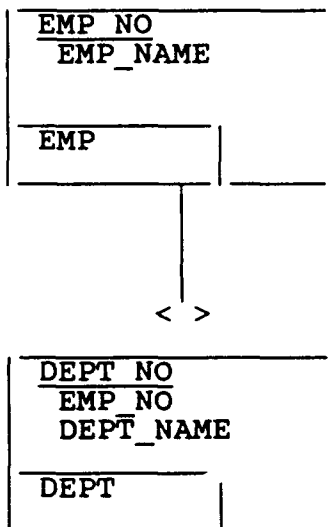
| | |
|----------|--|
| EMP NO | |
| EMP_NAME | |
| DEPT_NO | |

An entity class can have several horizontal partitions. Each record type maps to the entity class.

6.1.1.3 Joins

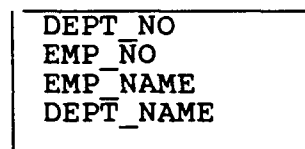
If some data fields in a record type map to attribute use classes in one entity class and others map to those in another, the two entity classes must be combined to form that record type. This is done with a relational "join" operation, which concatenates the entity instances of one entity class with those of the other. The two entity classes must be directly related by a relation class so that their entity instances can be matched using the key class of the independent and the corresponding inherited attribute use class(es) of the dependent.

CONCEPTUAL SCHEMA



INTERNAL SCHEMA

DEPT_EMP



If the relation class cardinality is one-to-many, each independent entity instance is concatenated with each entity instance that is dependent on it. In the first example in Figure 6-2, each PO-HEADER instance is formed by concatenating a Vendor instance with a PO instance based on identical values in Vendor No. If a Vendor instance has no dependent PO instances, it is not represented by a PO-HEADER instance. This produces one record instance for each instance in the dependent entity class, so the mapping must be to that entity class. If the mapping was to the independent entity class, i.e., if there was one record instance for each Vendor instance, P. O. NO. and any other attribute use classes from the P. O. entity class could occur multiple times in each record instance. Since a relational join cannot form record instances with repeating data fields, this situation is prohibited.

If the relation class cardinality is one-to-zero-or-one, the mapping can be to either the independent or the dependent entity class because neither can cause a repeating data field.

The second and third examples in Figure 6-2 show these two situations. In the second, there is one BUYER record instance for an employee who is not a buyer. In the third example, there is one EMP-MAST instance for each Employee instance. If an employee is not married, the SPOUSE-NAME data field in the record instance for that employee is null.

CONCEPTUAL SCHEMA

INTERNAL SCHEMA

ONE-TO-MANY-RELATION CLASS:

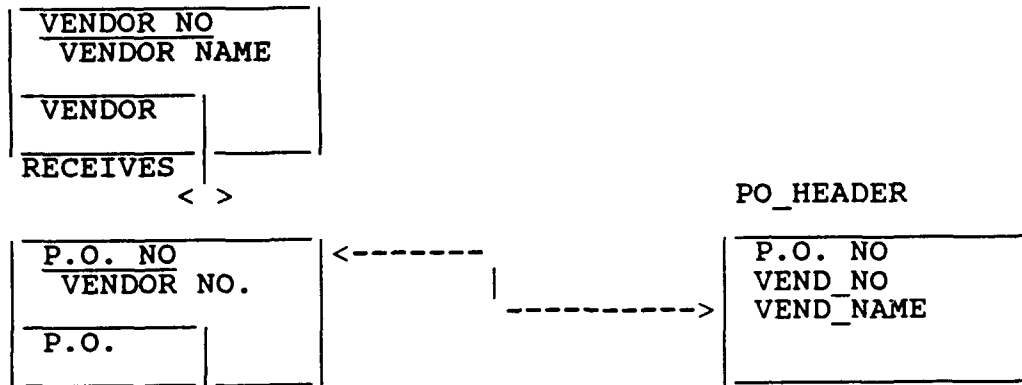


Figure 6-2. Join Examples

CONCEPTUAL SCHEMA

INTERNAL SCHEMA

ONE-TO-ZERO-OR-ONE RELATION CLASS:

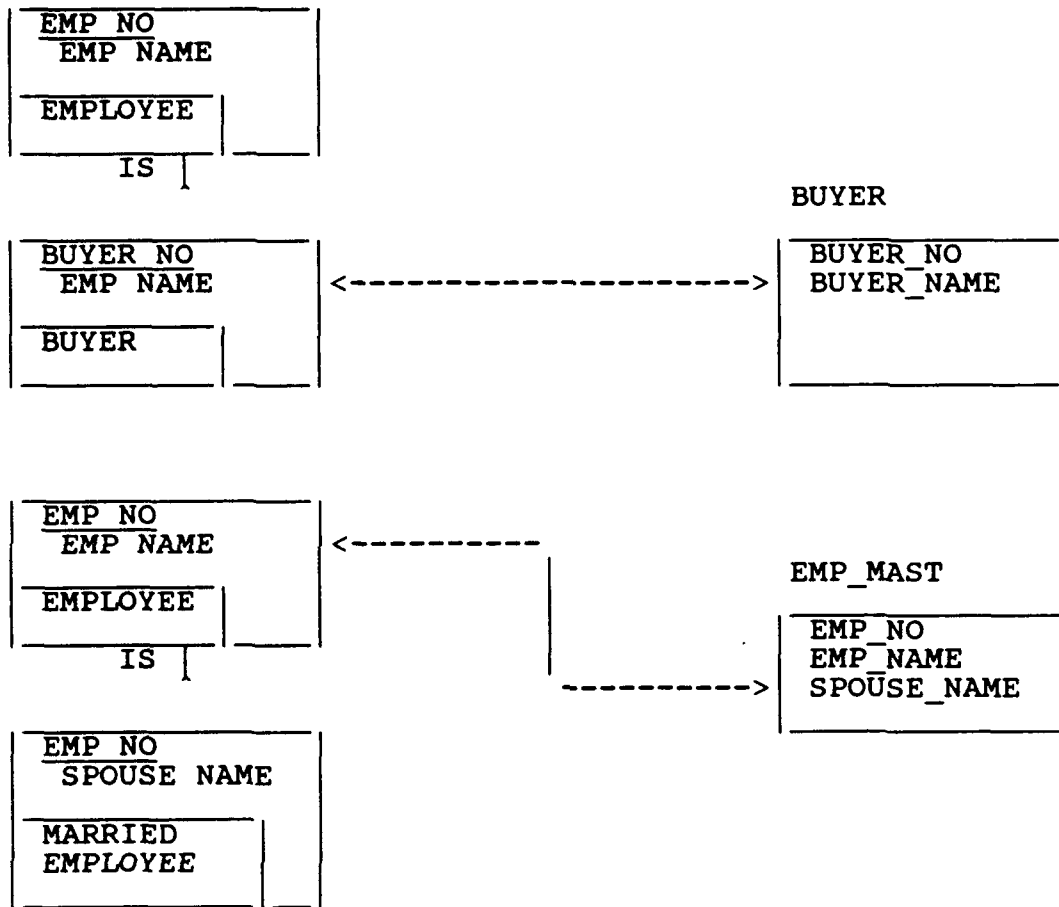


Figure 6-2. Join Examples (Continued.)

If a record type has data fields that map to attribute use classes in several entity classes, they must all be combined to form the record type. This is done with a series of the join operations described above, each of which combines two of the entity classes. All of the entity classes must be inter-related such that they form one of the following (See Figure 6- 3):

1. A regular hierarchy, i.e., a structure in which:
 - o One entity class, called the apex, is not dependent on any of the others (e.g., EC1)
 - o Every other entity class is dependent on exactly one entity class (not necessarily the same one for all)
 - o Every relation class cardinality is one-to-zero-or-one
2. A confluent hierarchy (an upside-down hierarchy), i.e., a structure in which:
 - o One entity class, called the apex, has none of the others dependent on it (e.g., EC14)
 - o Every other entity class has exactly one entity class dependent on it (not necessarily the same one for all)
 - o Any specific relation class cardinality is permitted
3. A combination of:
 - o One confluent hierarchy and
 - o One or more regular hierarchies, each of whose apex entity classes are also in the confluent hierarchy (e.g., EC15, EC20, and EC25).

Each hierarchy is called a join structure. As shown in the examples in Figure 6-3, the record type must map to the apex entity class of the regular or confluent hierarchy. If a combination of hierarchies exists, the mapping must be to the apex of the confluent hierarchy.

REGULAR HIERARCHY: CONCEPTUAL SCHEMA

INTERNAL SCHEMA

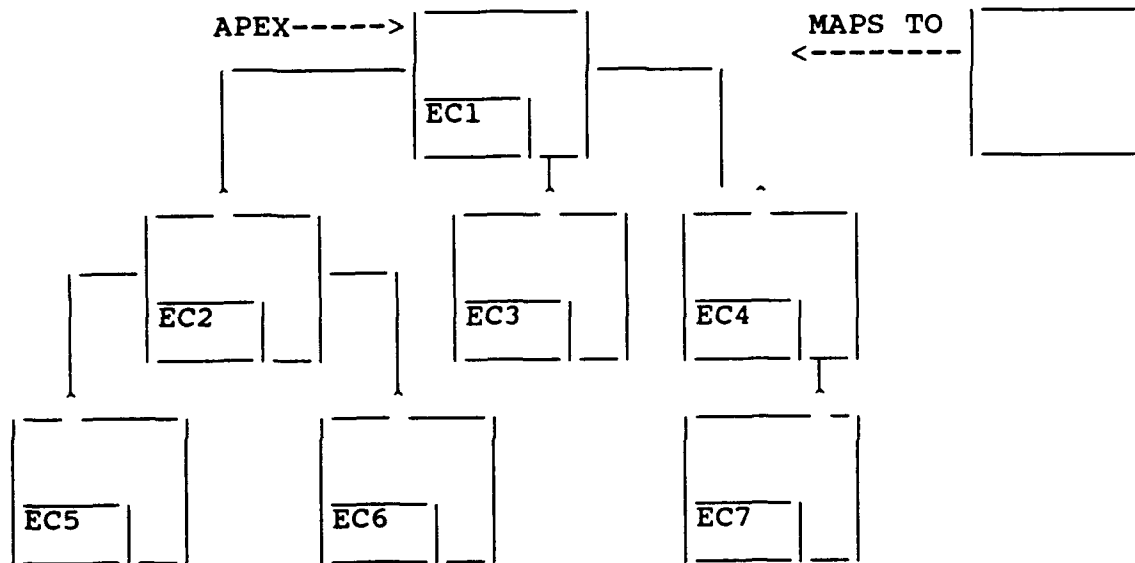


Figure 6-3. Join Structures

CONFLUENT HIERARCHY: CONCEPTUAL SCHEMA

INTERNAL SCHEMA

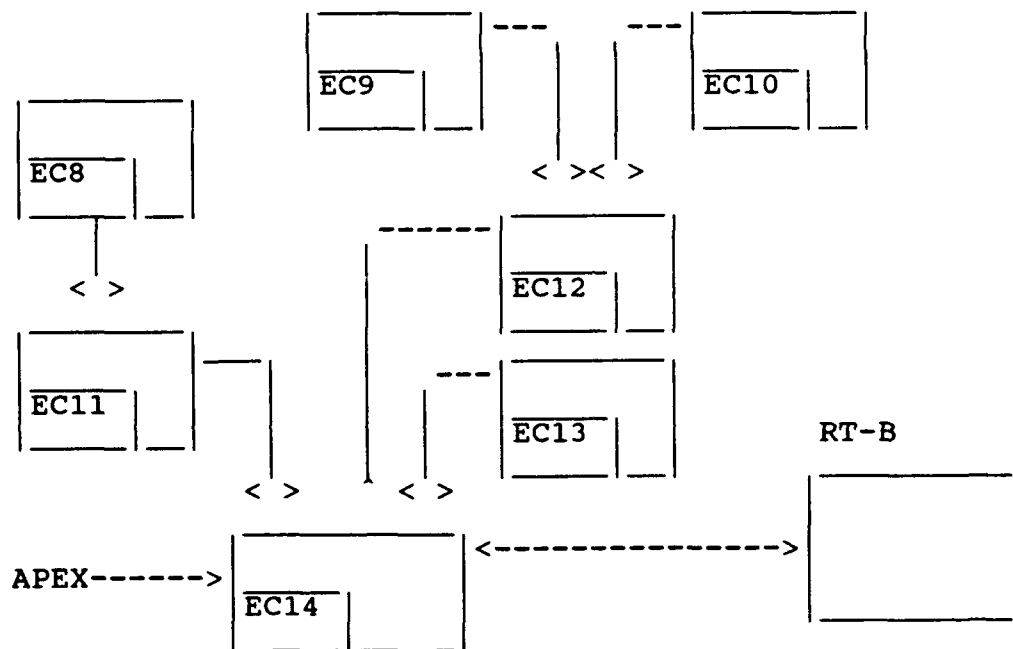


Figure 6-3. Join Structures (Continued)

COMBINATION: CONCEPTUAL SCHEMA

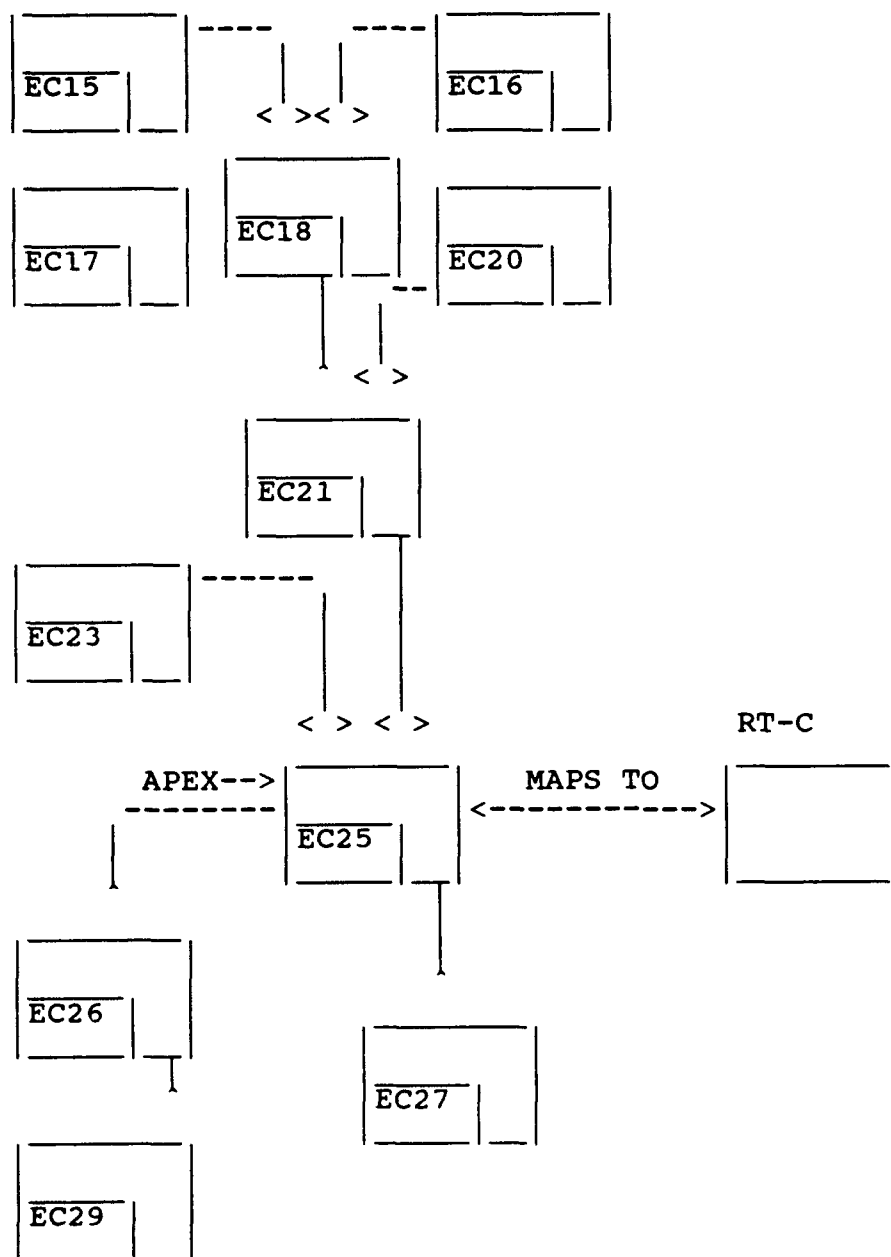


Figure 6-3. Join Structures (Continued)

6.1.1.4 Unions

A record type can map to two or more entity classes. This is the case when there is one record instance for each entity instance in one entity class and one for each in another. In the example below, each RESUPPLY record instance corresponds to either one Shop Order instance or one P. O. Item instance.

The creation of this sort of record type involves the use of the relational "union" operation. This allows the instances from both entity classes to be treated as if they were all the same kind of entity instances. Each data field can map to an attribute use class in each entity class, but that is not required. A data field can map to an attribute use class in one entity class without mapping to one in another. In the example above, these mappings are:

| <u>RESUPPLY</u> | | <u>SHOP ORDER</u> | | <u>P.O. LINE</u> |
|-----------------|---------|-------------------|-----|------------------|
| ORDER-NO | maps to | S.O. No. | and | P.O. No. |
| LINE-NO | maps to | | | Line No. |
| PART-NO | maps to | Part No. | and | Part No. |
| QUANTITY | maps to | S.O. Qty. | and | P.O. Qty. |
| AVAIL-DATE | maps to | Finish Date | and | Due Date |

LINE-NO does not map to an attribute use class in the Shop Order entity class. Consequently, each record instance that corresponds to an instance of that entity class is null in that data field.

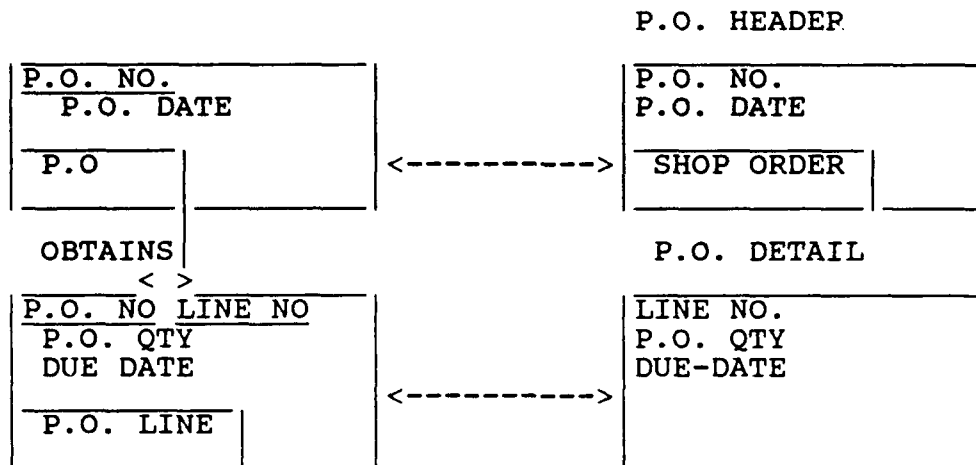
Usually, the entity classes involved in a union are directly related, although this is not a requirement.

6.1.1.5 Phantoms

Some DBMSs discourage the creation of data fields that would map to inherited attribute use classes. In the example below, P.O. No. in the P.O. Line entity class has no corresponding data field in the PO-DETAIL record type. Instead, when the purchase order number for an instance of that record type is needed, the one in the related PO-HEADER record instance is used.

CONCEPTUAL SCHEMA

INTERNAL SCHEMA



In this example, the P.O. No. in the P.O. Line entity class is called a "phantom" attribute use class because values for it can be retrieved from the database even though it does not map to any data field.

6.1.1.6 Duplications (Replications vs Redundancy)

Data duplication exists when the value in an attribute use class for a particular entity instance is stored in two or more data fields. In general, this is when an attribute use class maps to more than one data field. However, there are exceptions. When an entity class is horizontally partitioned, some or all of its attribute use classes map to more than one data field. If the partitions do not overlap though, i.e., if each entity instance corresponds to only one record instance, then each value is stored only once. Then, there is no data duplication. To summarize, data duplication exists when an attribute use class maps to two or more data fields unless all of those mappings result from a non-overlapping horizontal partition.

There are two types of data duplication:

Data Redundancy: The values in one of the data fields to which an attribute use class maps are not kept synchronized with those in another to which it maps.

Data Replication: The values in one of the data fields to which an attribute use class maps are updated and controlled to be kept synchronized with those in another to which it maps. Data replication may be used for performance purposes or for

purposes of joining across
physical record (but not for
joining entity classes).

As indicated by these definitions, data replication can be useful, but data redundancy is always undesirable. With data replication, updates to those multiple copies are controlled from a single source. The multiple copies are kept synchronized such that they reflect the same history of updates. By contrast, with data redundancy, updates to the multiple copies can be controlled by multiple sources, for example, by different applications. The result is that the copies may reflect different histories of updates and carry different values. When a user accesses redundant data, there can be no guarantee of the integrity or quality of that data. Depending on which copy is accessed, the user may receive very different results.

Whenever an attribute use class maps to more than one data field, the CDM Administrator must specify which is the "preferred copy." This is the data field that the CDM Processor will use for retrievals and qualifications in all NDML requests, regardless of application. The others will be used for joining across physical records when necessary. If an entity class has been horizontally partitioned, there should be a preferred copy designated in each partition. If a particular application wishes to use other than the preferred copy, it must bypass the CDM Processor and access that data field directly. It cannot use NDML for the request.

The CDM Processor treats all duplication as data replication; it must consider the values in all data fields to be synchronized. If, in fact, some duplication is really data redundancy, improper physical record joining may be performed, resulting in spurious responses.

6.1.1.7 Complex Mappings

A datafield to attribute use class mapping exists if they both represent the same "real-world" thing. Sometimes a datafield to attribute use class mapping can only be obtained by the programmatic manipulation of datafields or a record and attributes. Using the complex mapping example in Figure 6-26, the datafield BILL_TO_COUNTRY_ZIP is derived from the transformation of the two attributes BILL_TO_COUNTRY and BILL_TO_ZIP. A program is written in either COBOL or FORTRAN to perform this transformation. Likewise a program is written to process the datafield BILL_TO_COUNTRY_ZIP and break it down into the two attributes BILL_TO_COUNTRY and BILL_TO_ZIP. These programs provide CS to IS mappings and are called complex mapping algorithms.

6.1.2 CS-IS Mapping Modeling Forms

The following forms were developed to assist the CDM Administrator in determining the mappings between internal and conceptual schema objects. Most DBMSs provide a language for defining databases, for example, Data Definition Language (DDL). These NDDL statements for each database are referenced to compose the NDDL statements to define these internal schema objects to

the CDM (see Subsection 6.2, "Loading the Initial Internal Schema Objects"). Consequently, no internal schema modeling forms are needed. The following forms are used to model the mappings between internal schemas and the conceptual schema objects:

1. Record Type/Entity Class Mapping Form
2. Record Type Join Structure Diagram
3. Data Field/Attribute Use Class Mapping Form
4. Set Type/Relation Class Mapping Form

What follows is a detailed description and two samples (one blank and one filled in) of each of the Internal Schema Modeling Forms.

Record Type/Entity Class Mapping Form

Purpose: To provide a single source of information about the mappings between record types and entity classes.

Instructions:

Fill in one or more pages for each database. List each entity class to which record type maps. Refer to Figures 6-4 and 6-5.

| <u>Form Area</u> | <u>Explanation</u> |
|----------------------|--|
| 1. Database Name | Unique 30 character name assigned to the database by the CDMA. |
| 2. Record Type ID | Name or code that the DBMS uses to identify the record type. |
| 3. Entity Class Name | Name of the entity class to which the record type maps. |

Record Type Join Structure Diagram

Purpose: To provide a single source of information about the join structures for a record type.

Instructions:

Fill in one page for each record type that involves joining two or more entity classes. Refer to Figures 6-6 and 6-7.

| <u>Form Area</u> | <u>Explanation</u> |
|-------------------|--|
| 1. Database Name | Unique 30 character name assigned to the database by the CDMA. |
| 2. Record Type ID | Name or code that the DBMS uses to identify the record type. |

3. (Diagram Area)

Depiction of the entity classes
and relation classes that make up
the join structure.

| | | | | | | |
|--------------------------------|----------|-------|------------------|--------|------|---------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | | | | |
| | | | | | | |
| | | | | | | |
| NOTES: 1 2 3 4 5 6 7 8 9 10 | | | DRAFT | | | |
| | | | RECOM- MENDED | | | |
| | | | PUBLI- CATION | | | |

| DATABASE NAME | RECORD TYPE ID | ENTITY CLASS NAME |
|--|----------------|-------------------|
| 1 | 2 | 3 |
| <div style="display: flex; justify-content: space-between;"> <div>NODE: 5</div> <div>TITLE: RECORD TYPE ENTITY CLASS MAPPING</div> <div>NUMBER:</div> </div> | | |

Figure 6-4. Record Type/Entity Class Mapping Form

| | | | | | | |
|----------|----------|-------------------------|--------------|--------------|------|---------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | | | | |
| | | | | DRAFT | | |
| | | | | RECOM-MENDED | | |
| | NOTES: | 1 2 3 4 5 6 7 8 9 10 | PUBLI-CATION | | | |

| DATABASE NAME | RECORD TYPE ID | ENTITY CLASS NAME | |
|---------------|-----------------|---|--|
| PARTS DB | CP_PLAN_GROUP | OP EXEC PLAN GROUP | |
| PARTS DB | OP_PLAN_OP_PLAN | OP EXEC PLAN COMPONENT | |
| PARTS DB | OP_PLAN_PART | OP EXEC PLAN PART | |
| PARTS DB | PERSON | EMPLOYEE | |
| NODE: 5 | | TITLE: RECORD TYPE ENTITY CLASS MAPPING | |
| | | NUMBER: 2 | |

Figure 6-5. Record Type/Entity Class Mapping

| | | | | | | |
|--------------------------------|----------|-------|------------------|------------------|------|---------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | | | | |
| | | | | DRAFT | | |
| | | | | RECOM- MENDED | | |
| NOTES: 1 2 3 4 5 6 7 8 9 10 | | | PUBLI- CATION | | | |

| | | | |
|----------------|---|-----------------|---|
| DATABASE NAME: | 1 | RECORD TYPE ID: | 2 |
| 3 | | | |

| | | |
|-------|--|---------|
| NODE: | TITLE: RECORD TYPE JOIN STRUCTURE DIAGRAM | NUMBER: |
|-------|--|---------|

Figure 6-6. Record Type Structure Diagram

| | | | | | | |
|----------|--------------------------------|---------------|------------------|--------|------|---------|
| USED AT: | AUTHOR: PROJECT: | DATE: REV: | WORKING DRAFT | READER | DATE | CONTEXT |
| | | | RECOM- MENDED | | | |
| | NOTES: 1 2 3 4 5 6 7 8 9 10 | | PUBLI- CATION | | | |

| | |
|-------------------------|------------------------------|
| DATABASE NAME: PARTS DB | RECORD TYPE ID: MATERIAL_REQ |
|-------------------------|------------------------------|


```

graph TD
    ITEM[ITEM] -- IS --> PRODUCT_ITEM[PRODUCT ITEM]
    PRODUCT_ITEM -.->|IS REFERENCED BY| REQ[PRODUCT ITEM REQ]
    MATERIAL[MATERIAL]
    
```

The diagram shows three record types: ITEM, PRODUCT ITEM, and MATERIAL. ITEM is connected to PRODUCT ITEM with a solid line labeled 'IS'. PRODUCT ITEM is connected to PRODUCT ITEM REQ with a dashed line labeled 'IS REFERENCED BY'. MATERIAL is shown as a separate record type. The relationship between PRODUCT ITEM and PRODUCT ITEM REQ is also indicated by a vertical line with '< >' symbols.

| | | |
|-------|--|---------------|
| NODE: | TITLE: RECORD TYPE JOIN STRUCTURE DIAGRAM | NUMBER: 34 |
|-------|--|---------------|

Figure 6-7. Record Type Join Structure Diagram Example

Data Field/Attribute Use Class Mapping Form

Purpose: To provide a single source of information about the mappings between data fields and attribute use classes.

Instructions:

Fill in one or more pages for each record type in a database. List each attribute use class to which each data field maps. Refer to Figures 6-8 and 6-9.

| <u>Form Area</u> | <u>Explanation</u> |
|------------------------------|--|
| 1. Database Name | Unique 30 character name assigned to the database by the CDMA. |
| 2. Record Type ID | Name or code that the DBMS uses to identify the record type. |
| 3. Data Field ID | Name or code that the DBMS uses to identify the data field. |
| 4. Entity Class Name | Name of the entity class that contains the attribute use class tag in the next column. |
| 5. Attribute Use Class class | Tag name of the attribute use to which the datafield maps. |

| | | | | | | |
|----------|----------|-------------------------|------------------|-----------------|------|---------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | | | | |
| | | | | DRAFT | | |
| | | | | RECOM- MENDE | | |
| | NOTES: | 1 2 3 4 5 6 7 8 9 10 | PUBLI- CATION | | | |

| | | | |
|------------------|---|-------------------------|---------|
| DATABASE NAME: 1 | | RECORD TYPE ID: 2 | |
| DATA FIELD ID | ENTITY CLASS NAME | ATTRIBUTE USE CLASS TAG | |
| 3 | 4 | 5 | |
| NODE: | TITLE: DATA FIELD/ATTRIBUTE USE CLASS MAPPING | | NUMBER: |

Figure 6-8. Data Field/Attribute Use Class Mapping

| | | | | | | |
|----------|--------------------------------|-------|------------------|--------|------|---------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | | | | |
| | | | | | | |
| | | | | | | |
| | NOTES: 1 2 3 4 5 6 7 8 9 10 | | DRAFT | | | |
| | | | RECOM- MENDED | | | |
| | | | PUBLI- CATION | | | |

| DATABASE NAME: PARTS DB | | RECORD TYPE ID: RESOURCE | |
|-------------------------|---|--------------------------|---------------|
| DATA FIELD ID | ENTITY CLASS NAME | ATTRIBUTE USE CLASS TAG | |
| RES_ID | MFG_RESOURCE | MFG AREA ID | |
| RES_STATUS | MFG_RESOURCE | MFG AREA STATUS | |
| RES_NEXT_PM_DATE | MFG_RESOURCE | NEXT PM DATE | |
| RES_NEXT_PMR | MFG_RESOURCE | NEXT REQ NO | |
| RES_NEXT_OPG | MFG_RESOURCE | NEXT GROUP NO | |
| RES_MAX_OPG | MFG_RESOURCE | MAX GROUP NO | |
| RES_TONNAGE_ RATING | MFG_RESOURCE | TONNAGE RATING | |
| RES_TYPE | MFG_RESOURCE | MFG AREA TYPE | |
| NODE: | TITLE: DATA FIELD/ATTRIBUTE USE CLASS MAPPING | | NUMBER: 20 |

Figure 6-9. Data Field/Attribute Use Class Mapping Example

Relation Class/Set Type Mapping Form

Purpose: To provide a single source of information about the mappings between CODASYL set types or IMS paths and relation classes.

Instructions:

Fill in one or more pages for each database. List all of the set types that map to relation classes and all of the record types that are members in each set type. Refer to Figures 6-10 and 6-11.

| <u>Form Area</u> | <u>Explanation</u> |
|----------------------------|--|
| 1. Database Name | Unique 30 character name assigned to the database by the CDMA. |
| 2. Set Type ID | Name that the DBMS uses to identify the set type. |
| 3. Member Record Type ID | Name that the DBMS uses to identify a record type that is a member in the set type. |
| 4. Independent Entity Name | Name of the entity class that is independent in the relation class to which the set type maps. |
| 5. Relation Class Label | Label of the relation class to which the set type maps. |
| 6. Dependent Entity Name | Name of the entity class that is dependent in the relation class to which the set type maps. |

| | | | | | | | |
|----------|--------------------------------|-------|---|------------------|--------|------|---------|
| USED AT: | AUTHOR: | DATE: | - | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | | DRAFT | | | |
| | | | | RECOM- MENDED | | | |
| | NOTES: 1 2 3 4 5 6 7 8 9 10 | | | PUBLI- CATION | | | |

| DB NAME | SET TYPE ID. | MEMBER RECORD TYPE ID | INDEPENDENT ENTITY NAME | RELATION CLASS LABEL | DEPENDENT ENTITY NAME |
|--|-----------------|--------------------------|----------------------------|----------------------------|-----------------------------|
| 1 | 2 | 3 | 4 | 5 | 6 |
| <div> <div>NODE:</div> <div>TITLE: SET TYPE/RELATION CLASS MAPPING</div> <div>NUMBER:</div> </div> | | | | | |

Figure 6-10. Set Type/Relation Class Mapping

| | | | | | | |
|----------|--------------------------------|------------------|------------------|--------|------|---------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER | DATE | CONTEXT |
| | PROJECT: | REV: | DRAFT | | | |
| | | | RECOM- MENDED | | | |
| | NOTES: 1 2 3 4 5 6 7 8 9 10 | PUBLI- CATION | | | | |

| DB NAME | SET TYPE ID. | MEMBER RECORD TYPE ID | INDEPENDENT ENTITY NAME | RELATION CLASS LABEL | DEPENDENT ENTITY NAME |
|----------|--|------------------------|-------------------------|----------------------|-----------------------|
| PARTS DB | CAN BE PERFORMED_BY | INORTHROP_ALT_RESOURCE | ASSEMB_OP | IS PERFORMED BY | ALTERNATE_RES |
| PARTS DB | CARRIED_OUT_BY | ALT_RESOURCE | DRAWING_OP | IS PERFORMED BY | ALTERNATE_RES |
| PARTS DB | CONTAINS | RESOURCE_RESOURCE | MFG_RESOURCE | HAS | OP_GROUP |
| PARTS DB | CONSISTS_OF | OPERATION | ASSEMB_DEPT | HAS | OP_GROUP |
| PARTS DB | CONTROLLED_BY | STOCK_AREA_ITEM | MFG_PART | IS | MFG_ITEM |
| PARTS DB | CONTROLS STORAGE | STOCK_AREA_ITEM | MFG_WAREHOUSE | STORES | MFG_ITEM |
| NODE: | TITLE: SET TYPE/RELATION CLASS MAPPING | | | NUMBER: ---- | |

Figure 6-11. Set Type/Relation Class Mapping Example

The following subsections describe the methodology the CDM Administrator must follow in order to determine the CS-IS mappings for relational and CODASYL databases.

6.1.2.1 Relational Database Modeling Forms

ORACLE and DB2 relational DBMSs represent data as relations or two dimensional tables. These tables consist of columns (i.e., attributes) and rows (i.e., tuples). Figure 6-12 is an illustration of a relational implementation of the conceptual data model. All non-specific relationships have been resolved. All keys have been migrated and no role names are used. This makes the mapping from the conceptual schema to a relational database very straightforward. When mapping to this relational DBMS:

- * each entity class maps to a corresponding table
- * each attribute of the entity maps to the corresponding column of the table
- * the key of each entity becomes the primary key in the corresponding table
- * a relationship is represented by foreign keys in the dependent entity

The modeling forms pertinent to relational databases are the Record Type/Entity Class Mapping Form and the Data Field/Attribute Use Class Mapping Form. Refer to Figures 6-4 and 6-8. The CDM Administrator determines the primary mapping for each table by determining what sort of "real-world" thing the table represents. Each instance of a record type within a table contains data about a specific person, place, object, etc., that is significant to the enterprise. With a relational DBMS, all of the instances of the same type are about the same sort of thing and map directly to an entity. Once the entity class in the conceptual schema that represents the same sort of thing as the table is determined, the information is recorded in the Record Type/Entity Class Mapping Form. Using Figure 6-12 as an example:

E1 maps to Supplier Table
E2 maps to Order Table
E3 maps to Line-Item Table
E4 maps to Quotation Table
E5 maps to Part Table

Next, the CDM Administrator uses the Data Field/Attribute Use Class Mapping Form to assist in determining the primary mappings for each column of the tables. A one-for-one mapping should always exist between the attributes of an entity and the columns (i.e. datafields) of its corresponding table. A table, however, could contain datafields that exist only to maintain a relationship. These datafields will not have a mapping.

A few datafields may not contain data about "real-world" things and exist for technical reasons only. Examples include record codes and record activity dates. Such datafields do not map to any attribute use classes and can be ignored. The way the CDM Administrator determines which attribute use class in

the conceptual schema represents the same sort of data as the datafield is by finding the attribute use class whose definition or migration path corresponds to the intent of the datafield. This mapping information is recorded on the Datafield/Attribute Use Class Mapping Form. The relationship from the conceptual schema in Figure 6-12 are represented by attributes within tables and map as follows:

| | | |
|-----|--------------------|------------------------|
| RT1 | maps to Supplier # | in the QUOTATION Table |
| RT2 | maps to Supplier # | in the ORDER Table |
| RT3 | maps to Supplier # | in the LINE-ITEM Table |
| RT4 | maps to Part # | in the QUOTATION Table |
| RT5 | maps to Part # | in the LINE-ITEM Table |

INTERNAL SCHEMA

SUPPLIER

| SUPPLIER-NO | SUPPLIER-NAME | BILL-TO-ADDRESS | SHIP-TO-ADDRESS |
|-------------|---------------|-----------------|-----------------|
| KEY | | | |

PART

| PART-NO | PART-NAME | PART-DESCRIPTION | QTY-ON-HAND |
|---------|-----------|------------------|-------------|
| KEY | | | |

QUOTATION

| SUPPLIER-NO | PART-NO | QUOTE-PRICE | LEAD-TIME |
|-------------|---------|-------------|-----------|
| KEY | | | |

ORDER

| ORDER-NO | SUPPLIER-NO | ORDER-DATE | DELIVERY-DATE |
|----------|-------------|------------|---------------|
| KEY | | | |

LINE-ITEM

| ORDER-NO | LINE-ITEM-NO | PART-NO | QUANTITY | PRICE |
|----------|--------------|---------|----------|-------|
| KEY | | | | |

Figure 6-12. Relational Implementation of the Conceptual Model

CONCEPTUAL SCHEMA

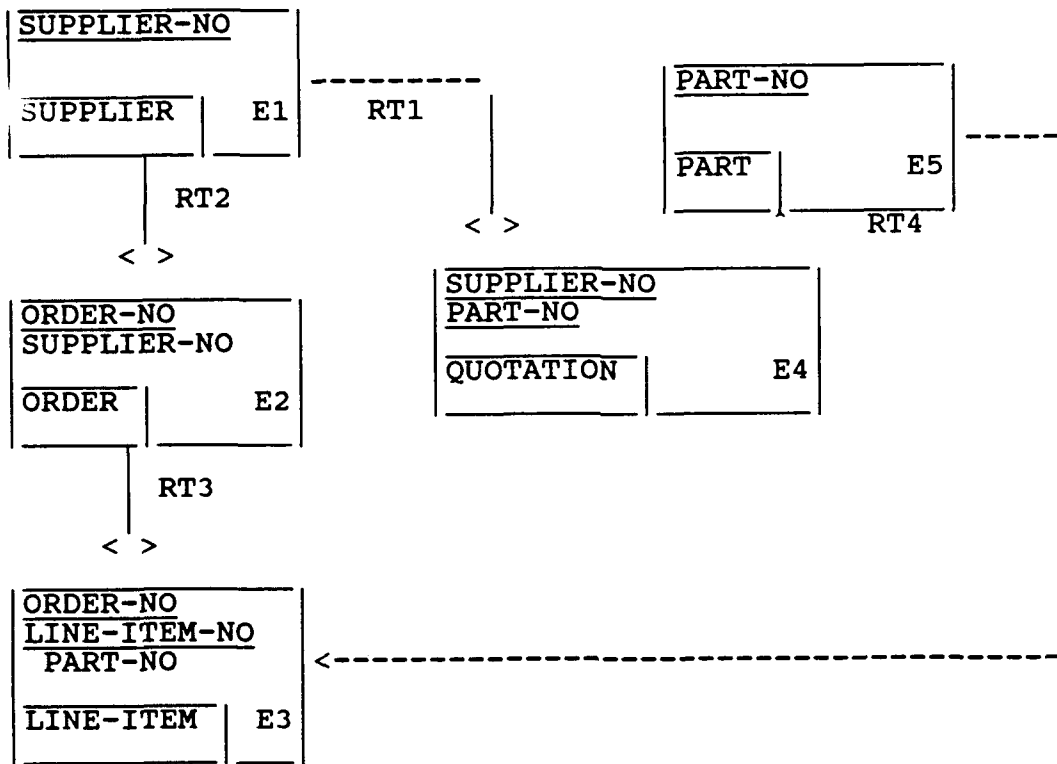


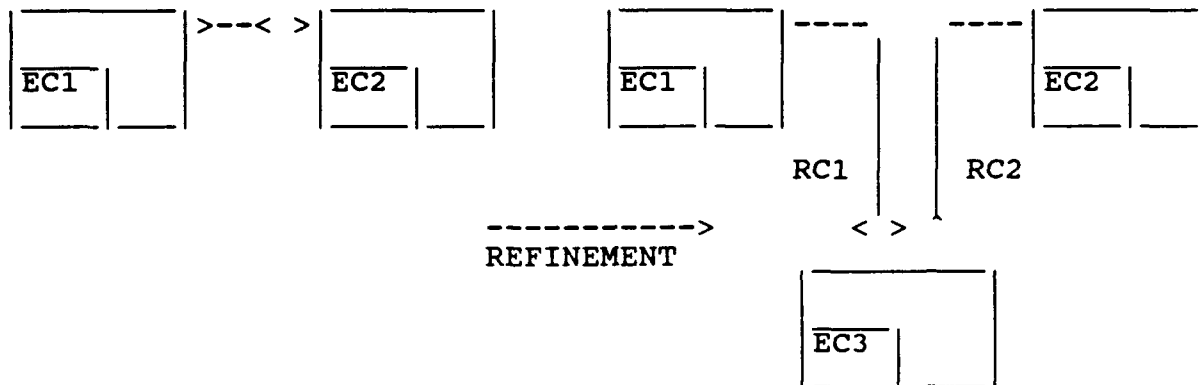
Figure 6-12. Relational Implementation of the Conceptual Model (Continued)

6.1.2.2 CODASYL Database Modeling Forms

The CODASYL VAX-11 DBMS offers two database design features that are not available in most others: multi-member set types and optional membership set types. The first is a single set type that has one owner record type and two or more member record types. Figure 6-13 is an example of a multi-member set. An owner instance can be associated with any number of instances of each type of member; they are not mutually exclusive. In essence, several logical relationships (relation classes) are combined into one physical relationship (set type). The CS-IS mapping for a multi-member set type involves:

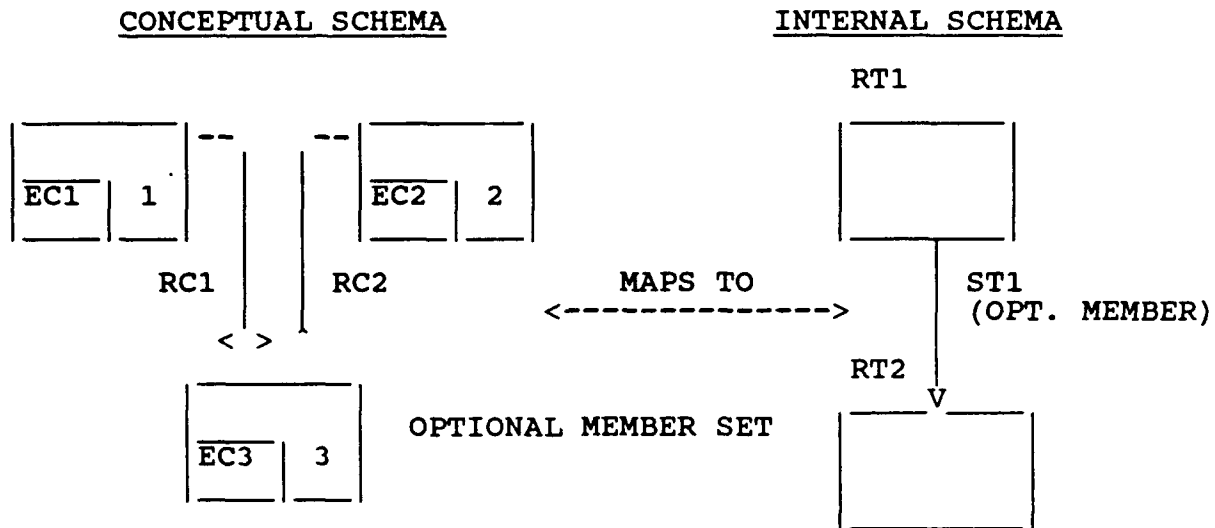
- * That the owner and member record types each have a primary mapping to a different entity class. Any of them can have secondary mappings also.
- * That the set type maps to several relation classes, one per member.
- * That the entity class that the owner maps to is independent in all of these relation classes.
- * That the entity class that a member maps to is dependent in one of these relation classes.

An optional membership set type is one in which an instance of the member record type is allowed to exist without being associated with an instance of the owner record type. This is in contrast with any other set type in which every member instance must be associated with an owner instance. An optional membership set type is equivalent to a non-specific relation class whose cardinality is zero or one-to-zero, one or many. Such a relation class is refined, as shown on the following page, before it is incorporated into the conceptual schema.



Consequently, the CS-IS mapping for an optional membership set type involves the following:

- * The owner record type has a primary mapping to one entity class, and the member record type has a primary mapping to another, and a secondary mapping to a third. Either one can have additional secondary mappings also.
- * The set type maps to a one-to-many relation class.
- * The entity class that the owner maps to is independent in that relation class.
- * The primary and secondary entity classes for the member are independent and dependent, respectively, in a one-to-zero or one, relation class, that is, a join linkage for the member.



All the modeling forms are pertinent when mapping VAX-11 databases to the conceptual schema. Refer to Figures 6-4, 6-6, 6-8 and 6-10. First the CDM Administrator determines the primary mapping for each record type. Usually, it is easier to map the record types that are not members in any set types first. Those that are set type members should not be mapped until all of their owner record types have been mapped. The CDM Administrator determines what sort of "real-world" thing the record type represents. Each instance of a record type contains data about a specific person, place, object, etc., that is significant to the enterprise. Usually, all of the instances of the same type are about the same sort of thing. This is not always the case, however. Referring to the figure in Subsection 6.1.1.4, an instance of the RE-SUPPLY-ORDER record type could represent either an order to the production department to make a certain quantity of parts, that is, a manufacturing order, or an order to a vendor to furnish a certain quantity of parts, a purchase order, for example. This is similar to defining an entity class. The datafields in a record type, especially those that uniquely identify its instances and the set types that it participates in, especially as a member, can all be useful in determining what the record type represents.

The CDM Administrator determines which entity class in the conceptual schema represents the same sort of thing as the record type by finding the entity class whose definition corresponds to the intent of the record type. Comparing the key classes, attribute use classes, and relation classes of the entity classes to the keys, datafields and set types of the record types can be helpful also. If the record type represents several sorts of things, it will map to several entity classes, one for each sort of thing. (See Subsection 6.1.1.4, "Unions"). If none of the entity classes represent what the record type does, either the record type exists only to improve database performance or the conceptual schema must be expanded (see Subsection 4.3). Once the entity class to which the record type

maps is determined, the information is recorded on the Record Type/Entity Class Mapping Form. Using Figure 4.3 as an example:

RT1 maps to EC1
RT2 maps to EC2
RT3 maps to EC3
RT4 maps to EC4
ST1 maps to RC1, RC2 and RC3

A few record types do not represent "real-world" things; they exist to improve database performance. Examples include SYSTEM-OWNER and entry points. Such record types do not map to entity classes and can be ignored.

Next, the CDM Administrator uses the Data Field/Attribute Use Class Mapping Form to assist in determining the primary mappings for each datafield. The content of the datafield is analyzed to result in what sort of data about "real-world" things it represents. If the record type that contains the datafield represents more than one sort of thing, i.e., it has more than one mapping, the datafield may contain several data types, all of these must be identified.

A few datafields may not contain data about "real-world" things and exist for technical reasons only. Examples include record codes and record activity dates. Such datafields do not map to any attribute use classes and can be ignored.

The way the CDM Administrator determines which attribute use class in the conceptual schema represents the same sort of data as the datafield is by finding the attribute use class whose definition or migration path corresponds to the intent of the datafield. The first place to look is the entity class to which the record type maps. If the record type maps to more than one entity class, the datafield may map to an attribute use class in each. The value in the datafield in each instance of the record type must be the same as the one in the attribute use class in the corresponding instance of the entity class. If two or more inherited attribute use classes that come from the same owned attribute use class have identical values in every entity instance, the datafield may map to some or all of them.

If none of the attribute use classes in the mapped-to-entity classes correspond to the datafield, the next places to look are the entity classes that are related to those entity classes. Again, the value in each record instance must be the same as the value in the corresponding entity instance. If the attribute use class is not in any of these entity classes, the search must be widened to include the entity classes that are related to them. This continues until the proper attribute use class is found or until it is determined that a new attribute class must be added to the conceptual schema. (See Subsection 4.3). This mapping information is recorded on the Datafield/Attribute Use Class Mapping Form.

The CDM Administrator's next step is to determine if any secondary mappings are needed for each datafield by finding the datafields in the record type that map to attribute use classes

that are not in the entity class to which the record type maps. This can be done by comparing the entity class names entered on the Data Field/Attribute Use Class Mapping Forms for the record types to those that are entered on the record type's record type/entity class mapping form. If an entity class name is on the first form but not on the second then that entity class must be joined with the one to which the record type maps.

Other entity classes might need to be identified to complete the join structure. The entity classes that must be joined to form the record type must form one or more join structures as described in Subsection 6.1.1.3. If the join structures are not contiguous, one or more additional joins may be needed. For example, if the record type, RT1, in Figure 6-14 maps to EC4 and involves joins with EC1 and EC3, it also must have a join with EC2. Without it, EC1 cannot be joined to the EC3-EC4 join result. The join must involve EC2 even though none of its attribute use classes map to datafields in the record type. Draw diagrams of these record types that involve joins on the Record Type Join Structure Diagram Form.

The final step for the CDM Administrator, when following this methodology, is to determine the mapping for each record set in which the record type is a member. What sort of relationship between "real-world" things the set type represents must be established. If the set type has more than one member record type, each must be considered separately. If either the owner or the member record type has no mapping to an entity class, the set type will have no mapping to a relation class, so it can be ignored.

The CDM Administrator determines which relation class in the conceptual schema represents the same sort of relationship as the set type. Usually, this is the relation class whose independent entity class maps to the owner record type and whose dependent entity class maps to the member record type. Fill out a line on a Set Type/Relation Class Mapping Form once the relation class to which the set type maps is determined. Use the optional member set type diagram above. As an example, the resulting primary, secondary and set mappings are:

| | |
|-----------------------------|--------|
| RT1 has a primary mapping | to EC1 |
| RT2 has a primary mapping | to EC2 |
| RT2 has a secondary mapping | to EC3 |
| ST1 has a mapping | to RC1 |

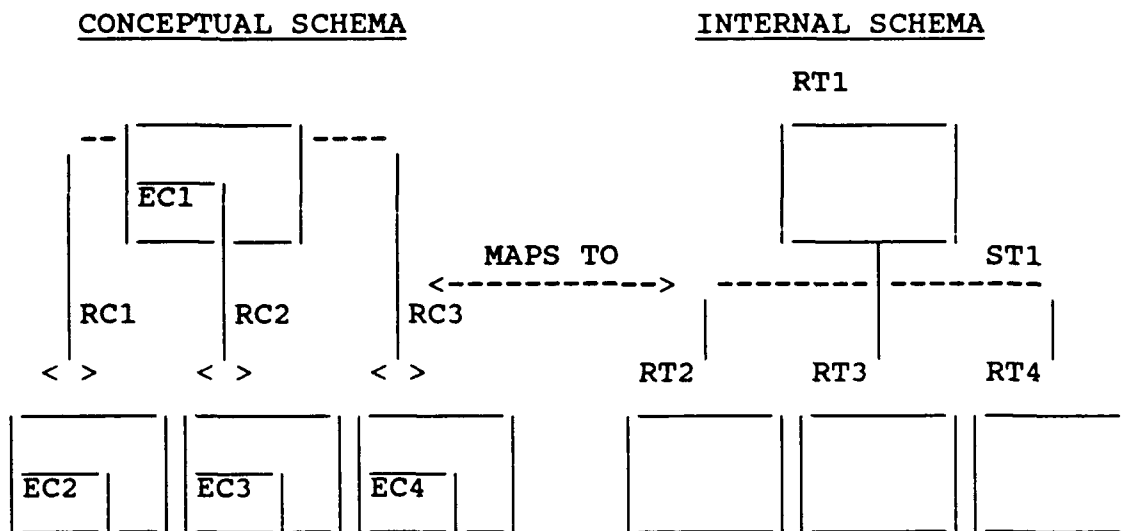
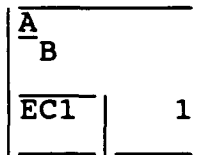
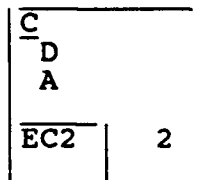


Figure 6-13. Multi-Member Set

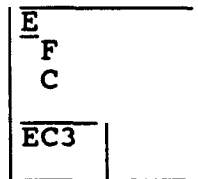
CONCEPTUAL SCHEMA



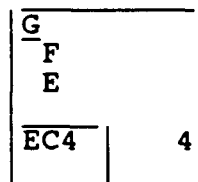
< >



< >



< >



INTERNAL SCHEMA

RT1

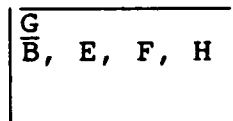


Figure 6-14. Incomplete Join Structure Example

6.2 Loading the Initial Internal Schema

The Internal Schema's objects are defined to the CDM in the following order:

User Defined data types (if different from the standard data type)

Database

Record Type

Datafield

Set/Path (for CODASYL and hierarchical databases)

The NDDL commands that define these common objects to the CDM are:

- a. ALTER DOMAIN
- b. DEFINE DATABASE
- c. DEFINE RECORD
- d. DEFINE SET

Before the internal schema is loaded to the CDM, the distributed database environment must be defined to the CDM. This means that the actual computer mainframes and the DBMSs that reside on those mainframes must be specified to the CDM. The NDDL commands that define this distributed environment to the CDM are:

- a. DEFINE DBMS
- b. DEFINE HOST

The Tables that are populated by these NDDL commands are in Figure 6-15.

6.2.1 Loading the Distributed Database Environment

A database management system is a set of computer programs that must be used to create and maintain a database. The DBMSs used to create and access the databases in the IISS must be defined to the CDM. This is accomplished with the DEFINE DBMS NDDL command. This command:

- * Gives the DBMS a name by which it is commonly known
- * Indicates which physical model the database is based (i.e., relational, hierarchical, network)

This NDDL command populates the IISS_DBMS CDM Table.

The optional ON HOST clause specifies which host computer the DBMS operates on. In order to use this clause, the host computer must have been defined to the CDM with the DEFINE HOST NDDL command. If this clause is specified, the DBMS_ON_HOST CDM Table is populated creating a cross reference between the DBMS and a host computer.

In an enterprise, data can reside on several computer mainframes. The IISS in conjunction with the DBMSs can access this data from a variety of computer sources. The CDM Administrator's task is to define the computer mainframes where the DBMSs operate and the data resides. This is accomplished with the DEFINE HOST NDDL command. A unique identification code is assigned to each host computer by the CDM Administrator. Using the DEFINE HOST NDDL command, the CDM assigns a unique HOST_NO to this identification code and populates the IISS_HOST CDM Table.

The optional WITH DBMS clause specifies which DBMSs operate on the computer just defined to the CDM. In order to use this clause, the DBMSs must have already been defined to the CDM with the DEFINE DBMS NDDL command. If this clause is specified, the DBMS_ON_HOST CDM Table is populated creating a cross reference between the host computer and the DBMSs.

The CDM Administrator can load descriptive text into the CDM describing the host(s) just defined, using the NDDL DESCRIBE command with an object identifier of "HOST".

6.2.2 Loading User-Defined data types

A domain can have several different styles for representing its values. These styles are defined with data types. A domain always has one standard data type that represents conceptual schema attributes (see Subsection 5.2.1, "Loading Domains"). As mentioned in this section, the format of the standard data type is limited to character, signed and unsigned formats. Many times it is necessary to describe the physical data storage representation of datafields differently than the standard data type's representation. Other data formats are float, integer, and packed. If a datafield differs in format from the standard data type's definition, another user-defined data type can be added to the domain with the NDDL ALTER DOMAIN command. Figure 6-16 illustrates the CDM tables that are populated by the NDDL ALTER DOMAIN command and the relationship between USER_DEF_DATA_TYPE and DATA_FIELD.

The ADD TYPE clause of the ALTER DOMAIN command adds an entry to the USER_DEF_DATA_TYPE CDM Table with the same DOMAIN_NO the CDM assigned when the Domain was originally created. The user-defined data type's name, type, size, and number of decimals are populated and the DATA_TYPE_IND is set to "USER". The relationship between this data type and the datafield is established when the actual database record is created.

The CDM Administrator can load descriptive text into the CDM describing the data type just defined, using the NDDL DESCRIBE command with an object identifier of "USERdata type".

6.2.3 Loading Databases

A database is a structurally interrelated collection of data. The interrelationships of this data are identified to database management systems for efficient and effective user access. The CDM Administrator has already defined the DBMSs and the host(s) they reside on when describing the distributed

environment to the CDM (see Subsection 6.2.1, "Loading the Distributed Database Environment"). Now the actual physical database models must be described to the CDM using the NDDL DEFINE DATABASE command. Besides providing the definition for the database, this command establishes the database named as current for the NDDL session. When the CDM Administrator wishes to define additional record types to an existing database, the database must be established as current for the NDDL session with an ALTER DATABASE NDDL command. Subsequent NDDL commands affecting internal schema objects will be performed against this current database.

The CDM Administrator can load descriptive text into the CDM describing these databases using the NDDL DESCRIBE command with the object identifier of "DATABASE".

6.2.3.1 Loading Relational Databases

The CDM Administrator loads ORACLE and DB2 database definitions into the CDM using the DEFINE ORACLE/DB2 DATABASE NDDL command. The DATA BASE CDM Table populated by this command is shown in Figure 6-17. The ORACLE or DB2 DBMS must be defined to the CDM prior to this command (see Subsection 6.2.1, "Loading the Distributed Database Environment"). When this command is issued, the CDM assigns a unique number to the database (DB ID) to identify the database named in the command. The optional clauses, STORES CHARACTER and STORES INTEGER allow the CDM Administrator to specify how null characters and integers should be stored in the database. If these optional clauses aren't used, null characters and integers are stored as the DBMS "null".

Another optional clause of the DEFINE DATABASE command is NTM DIRECTORY. This option allows the CDM Administrator to specify a two character code which is the prefix of a disk file directory. This directory is where the Network Transaction Manager of IISS will find the executable image for request processors generated by the precompiler. If this clause isn't used, the NTM Directory defaults to "GR".

A required clause when defining an ORACLE database is the WITH PASSWORD clause. The ORACLE DBMS restricts access to its databases unless a valid password is supplied. This clause stores the code that is a database password in the DB_PASSWORD CDM Table. This clause is not necessary for a DB2 database.

6.2.3.2 Loading CODASYL Databases

The CDM Administrator loads VAX-11 or other database definitions operated by CODASYL DBMSs into the CDM using the DEFINE VAX-11 DATABASE NDDL command. The DATA BASE CDM Table populated by this command is shown in Figure 6-18. The VAX-11 DBMS must be defined to the CDM prior to this command (see Subsection 6.2.1, "Loading the Distributed Database Environment"). When this command is issued, the CDM assigns a unique number to the database (DB ID) to identify the database named in the command. The optional clauses, STORES CHARACTER and STORES INTEGER allow the CDM Administrator to specify how null

characters and integers should be stored in the database. If these optional clauses are not used, null characters and integers are stored as zeros.

Another optional clause of the DEFINE DATABASE command is NTM DIRECTORY. This option allows the CDM Administrator to specify a two character code which is the prefix of a disk file directory. This directory is where the Network Transaction Manager of IISS will find the executable image for request processors generated by the precompiler. If this clause is not used, the NTM Directory defaults to "GR".

A required clause when defining a VAX-11 database is the SCHEMA clause. The schema, subschema and area names are given for the database with this clause and the SCHEMA_NAMES and DATA_BASE_AREA CDM Tables are populated with these names. The LOCATED AT clause is also required when defining a VAX-11 database to the CDM. It specifies the fully qualified VAX directory name where the VAX-11 database is stored. The DB_LOCATION column of the SCHEMA_NAMES CDM Table is populated with this clause. This location is used by the precompiler for generating the DB statement in VAX-11 CODASYL COBOL programs.

Database definition statements used by TOTAL DBMSs are loaded into the CDM using the DEFINE TOTAL DATABASE NDDL command. The clauses of this command that are pertinent to CODASYL databases are also applicable to TOTAL databases with the exception of the SCHEMA and LOCATED AT clause.

6.2.4 Loading Record Types and Data Fields

A record is a group of data values that are stored together as a unit in a database. A record type is the collection of all the records that represent the same kind of information (i.e., all the records that contain similar data values). A datafield is the portion of the record type where the actual data values can be stored. Because a datafield is a subset of a record type, both are defined to the CDM at the same time, with the DEFINE RECORD NDDL command. With this command the word RECORD, TABLE and SEGMENT can be used interchangeably depending on which name most appropriate for the DBMS' record being defined.

The CDM Administrator loads descriptive text into the CDM describing these new records and datafields using the NDDL DESCRIBE command and either the object identifier "RECORD" or "DATAFIELD".

6.2.4.1 Loading the Relational DBMS' Record Types

The CDM Administrator loads record type definitions for ORACLE and DB2 databases using the DEFINE TABLE NDDL command. If the current database has not been established for the NDDL session with a prior DEFINE or ALTER DATABASE NDDL command, it can be established with the optional OF DATABASE clause in this command. The RECORD_TYPE CDM Table in Figure 6-17 is populated by this command so far.

The datafields of an ORACLE and DB2 Table are called columns and the WITH COLUMNS clause provides the names of the datafields in the records. The optional LEVEL_NO clause if not specified defaults to "1". This clause need not be specified since ORACLE and DB2 DBMSs only support elementary datafields. An elementary datafield must have a data type name associated with it. The DATA TYPE clause specifies the data type name whose definition describes the physical internal data storage representation for the datafield. This clause establishes the relationship between the datafield being defined and the data type already created (see Subsection 6.2.2, "Loading User-Defined data types"). The ORACLE DBMS stores numeric data without decimals in INTEGER NDDL data type format and numeric data with decimals in PACKED NDDL data type format. Since the standard data type of a domain can only have CHARACTER, SIGNED and UNSIGNED NDDL data type formats, a new data type will have to be added for any datafields with numeric representations. The ORACLE DBMS stores character data in CHARACTER NDDL data type format.

The optional clause KNOWN can be omitted because its default is KNOWN. This clause specifies if the DBMS can address this datafield by name. If not, it is specified as UNKNOWN. ORACLE and DB2 DBMSs address datafields by name, therefore, each column must be KNOWN and the order of the columns specified in the record type is not important. The DATA_FIELD CDM Table in Figure 6-17 has been populated with the information specified thus far.

ORACLE and DB2 DBMSs do not support repeating fields, COBOL type indexes or redefinition of datafields. The CDM does not need knowledge of the record type's indexes in an ORACLE database. Therefore, all other optional clauses on the DEFINE TABLE NDDL command are not applicable.

6.2.4.2. Loading CODASYL DBMS' Record Types

The CDM Administrator loads record type definitions for VAX-11 databases using the DEFINE RECORD NDDL command. If the current database has not been established for the NDDL session with a prior DEFINE or ALTER DATABASE NDDL command, it can be established with the optional OF DATABASE clause in this command. The RECORD_TYPE CDM Table in Figure 6-18 is populated by this command so far.

The optional IN AREAS clause assigns the record type to a database area, if the database has been subdivided into areas (see Subsection 6.2.3.2, "Loading CODASYL Databases"). This subdivision is a technique for improving the efficiency of accessing record type instances. When a database is subdivided into database areas and this record type is assigned to a particular area with this clause, then these record type instances can be accessed by searching only the appropriate areas rather than the entire database. This clause populates the DB_AREA_ASSIGNMENT CDM Table in Figure 6-18.

The datafields of a VAX-11 record are called fields and the WITH FIELDS clause provides the names of the datafields in the records. using the actual physical record layouts of the VAX-11 records, the DEFINE RECORD NDDL command can be composed using all its optional clauses. The NDDL User's Manual gives an example of

using this physical record layout and translating it into the appropriate DEFINE RECORD NDDL command. All the NDDL clauses (i.e., LEVEL NO, OCCURS, INDEXED BY, REDEFINES, UNIQUE and DUPLICATE KEY) are pertinent to VAX-11 records because access is done on a record at a time basis.

The DEFINE RECORD NDDL command is also used to load record type definitions for TOTAL record types. All clauses pertain to TOTAL records since the application program may redefine the entire record in any way it sees fit. The exception is the IN AREAS clause. This clause does not apply to TOTAL record types because the AREA_ID is derived from the first four positions of the record name.

CODASYL DBMSs utilize the concept of record sets. A record set indicates that an instance of one record type "owns" some number of instances of another record type. Instances of the first are called "owners" and those of the second "members". This relationship between CODASYL record types is defined to the CDM with the DEFINE SET NDDL command. The RECORD_SET and SET_TYPE_MEMBER CDM Tables in Figure 6-18 are populated with this command.

TOTAL DBMSs also utilize the concept of record sets. The DEFINE SET command's optional clause, LINKED BY, is required when defining a TOTAL record set to the CDM. The datafield specified to link the records of the set must reside in the member record of the set. The DF_SET_LINKAGE CDM Table is populated with this clause.

The CDM Administrator can load descriptive text into the CDM describing the record sets just defined, using the NDDL DESCRIBE command with an object identifier of "SET".

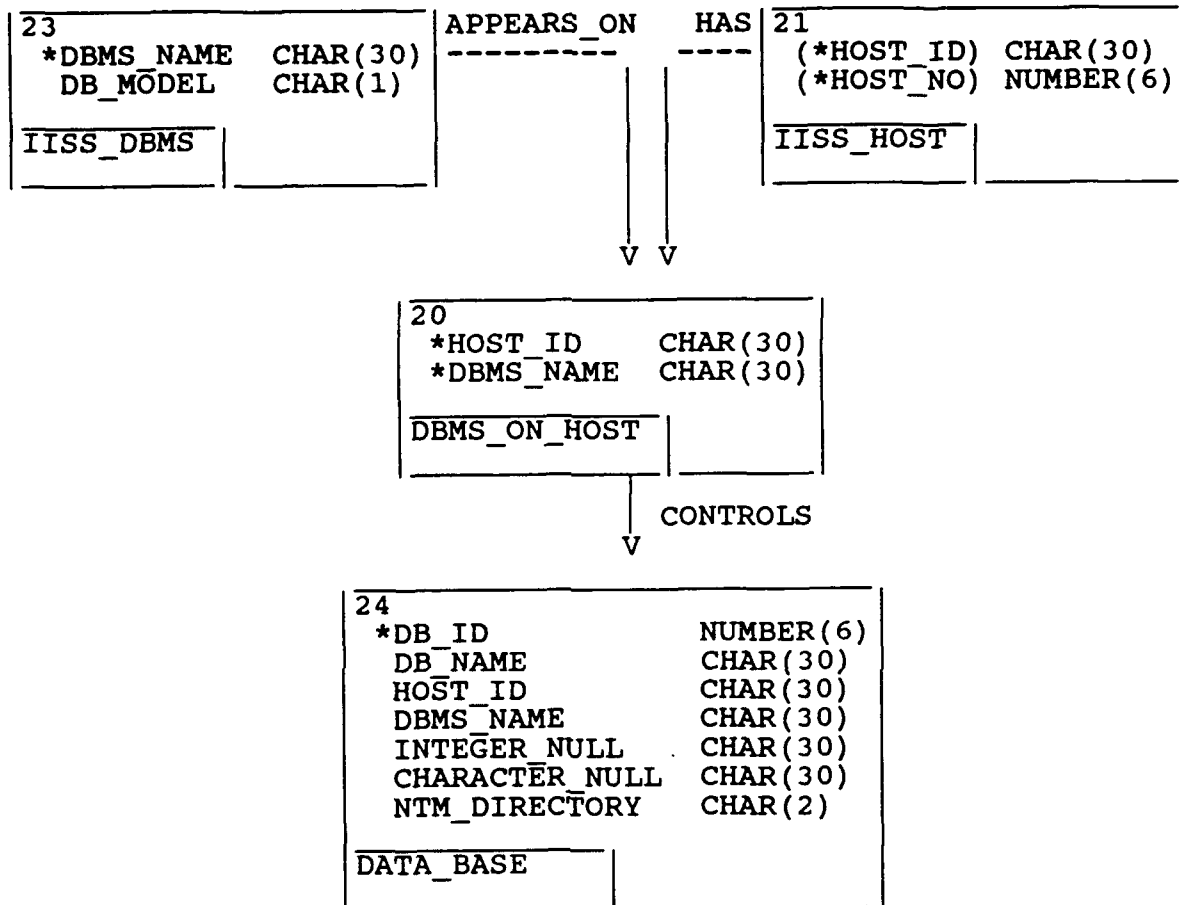


Figure 6-15. CDM Tables Distributed Data Bases

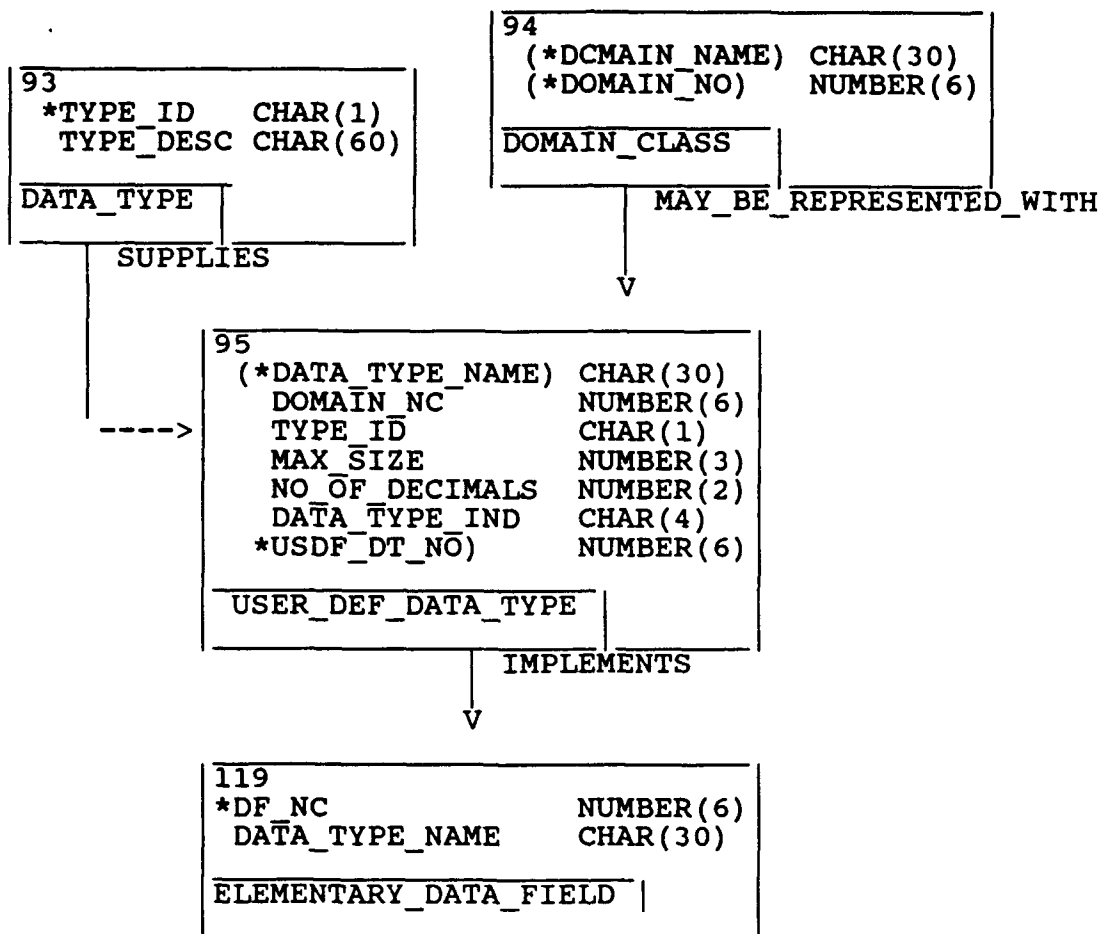


Figure 6-16. CDM Tables Domains and Data Types for Internal Schema

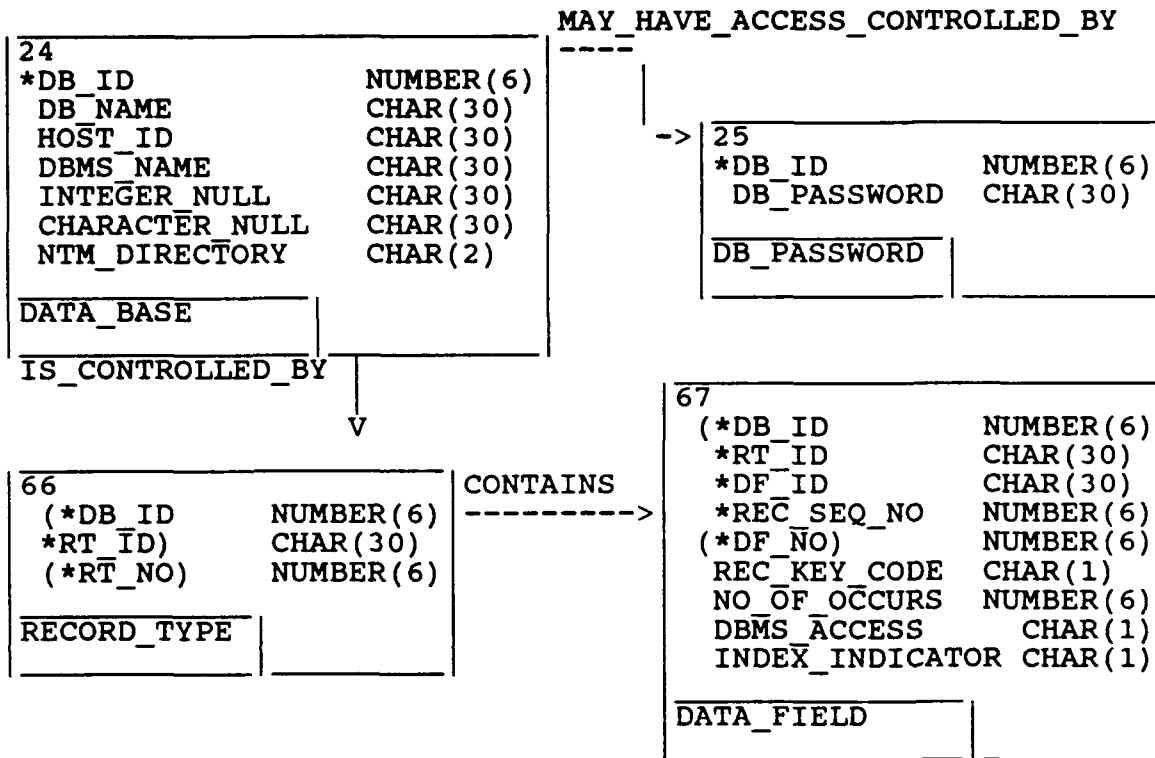


Figure 6-17. CDM Tables Relational Database Internal Schema

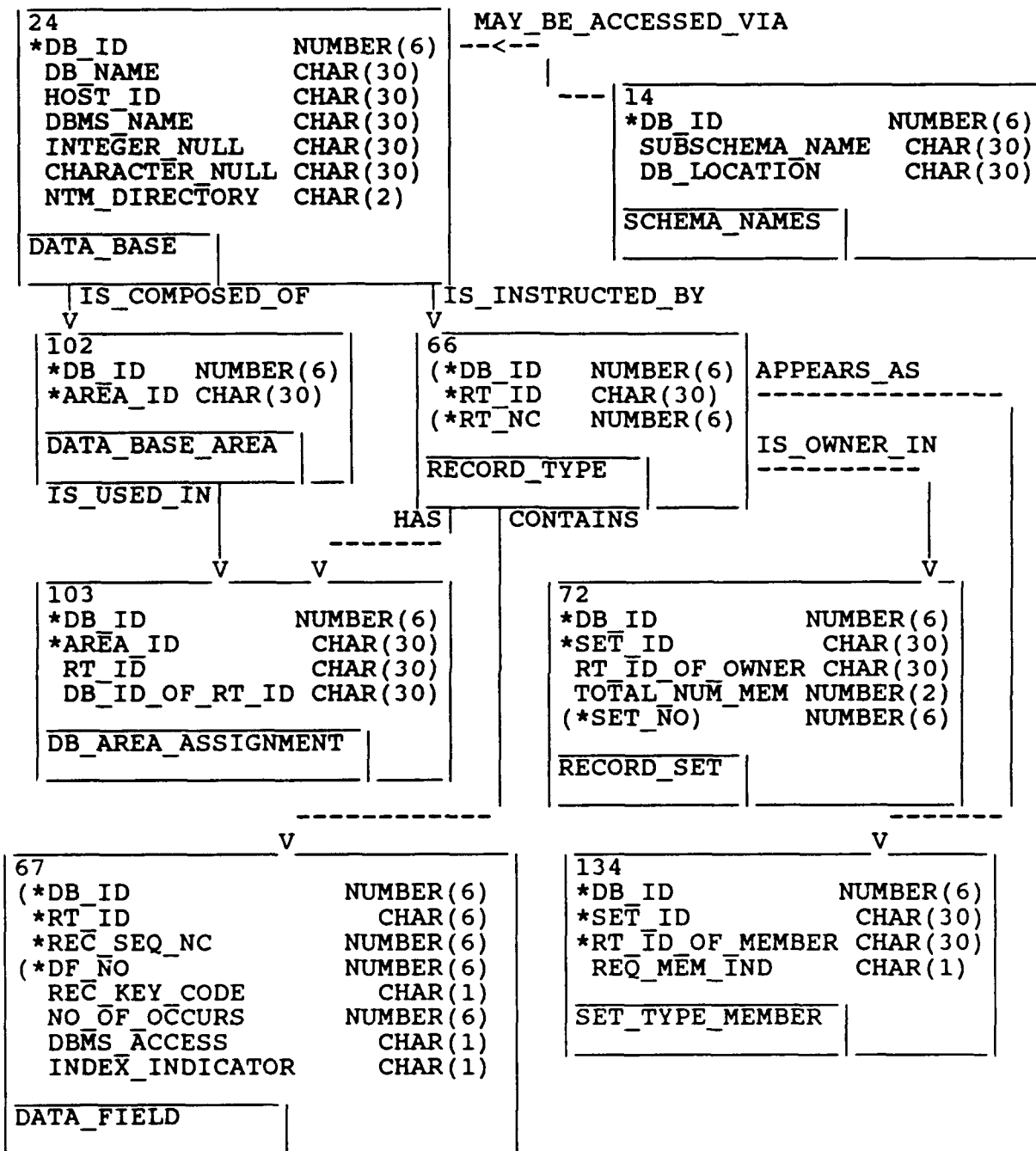


Figure 6-18 CODASYL Internal Schema

6.3 Loading the Initial CS-IS Mapping Definition

Once the CDM Administrator defines the internal schema to the CDM, the next task at hand is to load the mapping definitions of these internal schema objects to the appropriate conceptual schema objects. CS-IS Mapping definitions are loaded into the CDM in the following order:

1. Entity Class to Record Type
2. Attribute Use Class to Datafield or Set Values
3. Relation Class to Record Set Relationship
(for CODASYL databases)

Each of these mapping levels has a modeling form which was completed by the CDM Administrator, following the methodology detailed in Subsection 6.1.2, "CS-IS Mapping Modeling Forms". These modeling forms serve as source documents when defining the CS-IS mappings to the CDM. Different variations of the NDDL CREATE MAP command is used to load all levels of CS-IS mapping definitions into the CDM.

6.3.1 Loading CS to IS Mappings

The source document for entity class to record type mappings is the Record Type/Entity Class Mapping Form. Besides defining the CS-IS mapping to the CDM, the Create Map establishes the distributed update and retrieval rules for the entity class. The distributed retrieval rule is set to "allow" if the CDM Administrator wants the precompiler to generate retrieval subtransactions for secondary mappings of this entity in the event that:

- (1) the primary copy is on a different host than the one the NDML application program is to execute on
- (2) an active mapping exists to a secondary copy on the target host of the NDML application program

The distributed update rule is set to "allow" if the CDM Administrator wants the precompiler to generate update subtransactions for all distributed mappings (i.e., secondary or non-primary) of this entity. If the CDM Administrator chooses to allow distributed retrieval and disallows distributed update for the entity class being mapped in Figure 6-20, the resulting NDDL CREATE MAP command would be:

```
create map STOCK_AREA_LOCATION
to record PARTSDB.LOCATN
allow retrieval disallow update;
```

This command updates the EC_RT_MAPPING and DISTRIBUTED_RULES CDM Tables in Figure 6-19.

The source document for attribute use class to datafield mappings is the Datafield/Attribute Use Class Mapping Form. The CDM Administrator chooses what this CS-IS mappings' classification should be as well as its mapping category. The mapping classification is for documentation purposes only and

specifies which of the various types of replicated or duplicated data the particular mapping represents. The map category is used to indicate whether or not the CS-IS mapping for an attribute use class is to be used by the NDML precompiler when looking for valid versions or storage locations for the attribute use class. If the CDM Administrator chooses a mapping category of "active" and a mapping category of "original source" for the attribute use class in Figure 6-22, the resulting NDDL CREATE MAP command would be:

```
create map STOCK_AREA_LOCATION.STOCK_AREA_LOCATION_ID
      active original source for preference 1
      to field PARTSDB.LOCATN.LOC_ID;
```

The AUC_IS_MAPPING CDM Table in Figure 6-21 is populated by this command and the cross reference between the attribute use class and the datafield is established by the population of the PROJECT_DATA_FIELD CDM Table.

The Set Type/Relation Class Mapping Form is relevant to CODASYL VAX-11 databases and the source document for relation class to record relationship (i.e., set types) mappings. First the set must be defined to the CDM using the DEFINE SET NDDL command (see Subsection 6.2.4.2, "Loading CODASYL DBMS' Record Types). Once the set is defined to the CDM, its mapping to a relation class can be defined. Using the set type in the Set Type/Relation Class Mapping Form in Figure 6-23 as an example, the resulting CREATE MAP NDDL command is

```
create map STOCK_AREA IS_COMPOSED_OF STOCK_AREA_LOCATION.
      to set PARTSDB.PHYSICALLY_CONTROLS.LOCATN;
```

The RC_BASED_REC_SET CDM Table in Figure 6-21 is populated by this command.

Attribute use classes may also map to set types. This happens when certain attribute use classes can be represented in a database by a group of record sets rather than by a datafield. For example, Project Task Status might be represented by four PROJECT to TASK record sets called PENDING, IN-PROCESS, ON-HOLD and COMPLETED. An attribute use class record set mapping indicates that a particular record set corresponds to a particular attribute use class depending on its value. The attribute use class to set type variation of the CREATE MAP command populates the AUC_IS_MAPPING and AUC_ST_MAPPING CDM Tables in Figure 6-21.

6.3.2 Loading Record Unions

It has been mentioned when explaining the CS-IS mapping methodology that a record type may correspond to more than one entity class, therefore the record type will have more than one entity to record mapping. This is a relational union of those entity classes (see Subsection 6.1.1.4, "Unions"). Some instances of such a record type correspond to instances of one of the entity classes, others to those of another. The way to determine which record instances correspond to instances of each entity class is by a union discriminator.

A union discriminator is a specific value that, when found in a datafield, indicates which entity to record mapping should be used. Once the CDM Administrator determines that a record union exists, it can be defined to the CDM with the CREATE MAP command to map each entity to the record and to state the distributed rules, followed by the CREATE UNION NDDL command which names the record type and the entity classes that are unioned, along with the union discriminators for each entity. Using the depiction of the record union of entities DRAWING and SPEC in Figure 6-24 the resulting NDDL Command is:

```
create union of record PIOS.DWG_SPEC_REC
  to entity DRAWING when DS_REC_TYPE = "1"
  SPEC when DS_REC_TYPE = "2";
```

This command updates the ECRTUD Table in Figure 6-19.

6.3.3 Loading Horizontal Partitions

Some record types do not store all instances of an entity class. One is used to store some instances while another is used to store others. Each record type represents a fragment of the entity class. These fragments do not overlap (i.e., no entity class instance appears in more than one fragment. An entity class can be partitioned into any number of fragments, usually with each being in a different database or file. An example of this type of horizontal partition is in Figure 6-25 where the record type fragments reside in two different databases. However, this is not a requirement of horizontal partitions. Some or all fragments may be stored as different record types in the same database or file. A constraint statement defines each fragment (i.e., describes the conditions that must be met by each entity instance that is stored as a given record type).

The same entity may be partitioned more than once. For example, the entity books may be partitioned based on the attribute values of bookshop locations and based on attribute values in the publishing company names. These different partitions are specified by a partition number in the CREATE PARTITION NDDL command. This NDDL command updates the HORIZONTAL PART CDM Table in Figure 6-19. Using the illustration of the horizontal partition of entity PART in Figure 6-25 the resulting NDDL Command is

```
create partition 1 of entity PART
  to record MCMC.PART_RECORD
  PIOS.PART_INFO_REC;
```

6.3.4 Loading Transformational Algorithms

When a complex mapping exists between a datafield and an attribute use class, a software module can be invoked to provide the mapping. Examples of a complex mapping are: data transformations, unit of measure conversions, calculated fields, etc. A search or update software module has any number of attribute use classes or constants as its input parameters and any number of datafields or a record as output parameters. A

retrieval algorithm can be written to take as its input any number of datafields, constants or a record and any number of attribute use classes as its output parameters.

The NDDL DEFINE MODULE command defines a software module which, when invoked, performs the CS-IS or IS-CS mapping. Its input and output parameters, as well as their data types, are listed in this command. This command populated the CDM Tables SOFTWARE_MODULE and MODULE_PARAMETER in Figure 6-27.

The NDDL DEFINE ALGORITHM command defines to the CDM that the software module just defined is to be used as a complex mapping algorithm between:

- a. datafield(s) and attribute use class(es) or
- b. record and attribute use class(es)

The attribute use classes listed must exist in related entities. Any number of constant variables may be listed as input parameters to the algorithm. The same generic software module may be used to specify many complex mapping algorithms. The algorithm must specify in which direction the algorithm is transforming; e.g., conceptual to internal, specified as update or internal to conceptual, specified as retrieval.

The following NDDL commands are used in Figure 6-26:

```
DEFINE MODULE ADDRTRN IN COBOL
  PARAMETERS
    ADDRESS TYPE ADDRESS
    COUNTRY-PART TYPE COUNTRY
    ZIP-PART TYPE ZIP
    DIRECTION TYPE ALG DIRECTION
    RET-STATUS TYPE RET-STATUS ;

DEFINE ALGORITHM ADDRTRN      1 FOR UPDATE
  FOR PREFERENCE              1 USING PARAMETERS
  COUNTRY-PART FROM ATTRIBUTE CUSTOMER.BILL_TO_COUNTRY
  ZIP-PART FROM ATTRIBUTE CUSTOMER.BILL_TO_ZIP
  DIRECTION CONSTANT 'U'
  ADDRESS TO DATAFIELD PIOS.CUST.BILL_TO_COUNTRY_ZIP
  STATUS
;
```

The CDM Tables populated by this NDDL command are AUC_PARM, RT_PARM or DF_PARM, and CONST_PARM.

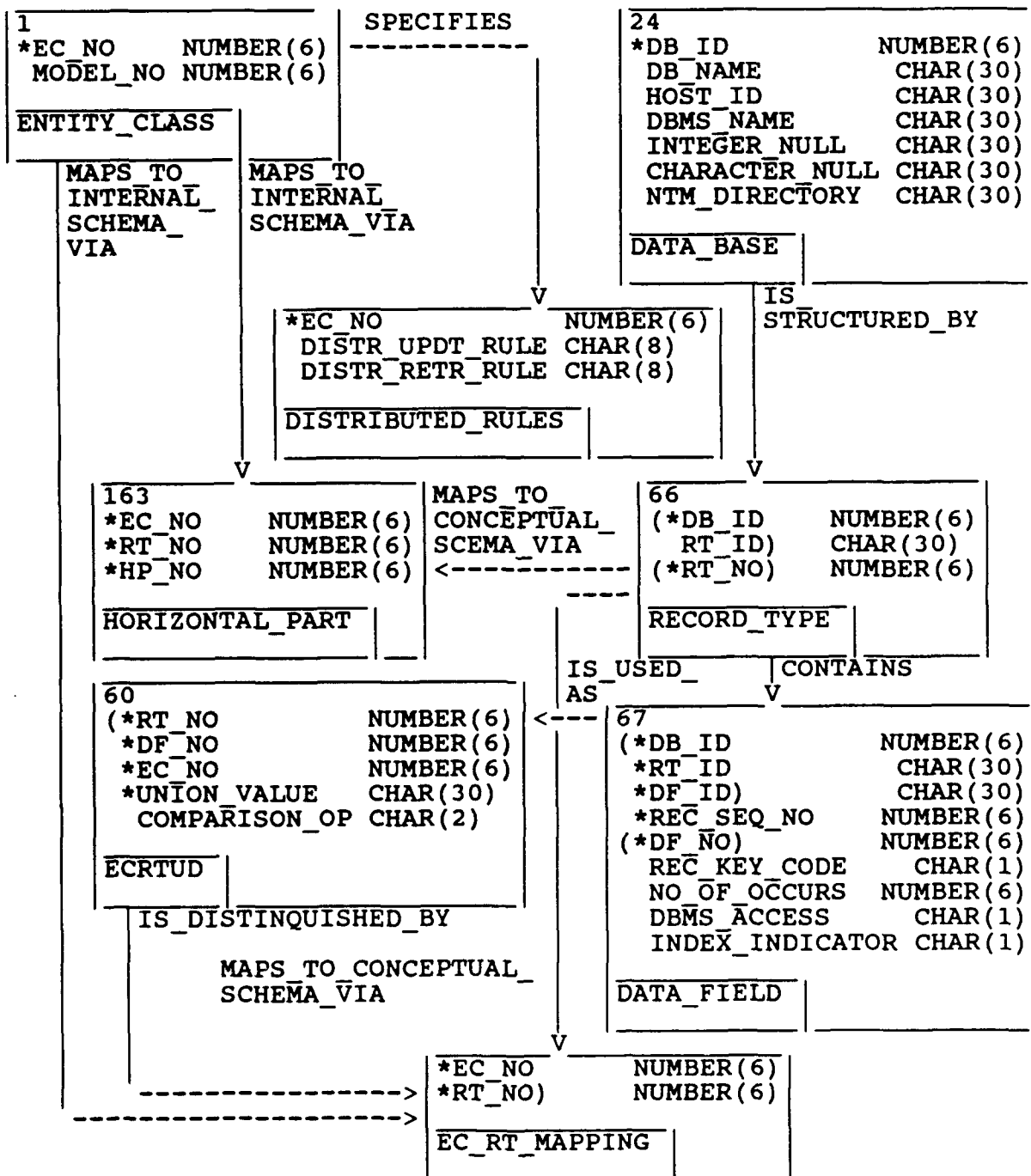


Figure 6-19. CS to IS Entity Mapping

| | | | | | |
|----------------------|----------|-------|--------------|-------------|----------|
| USED AT: | AUTHOR: | DATE: | WORKING | READER DATE | CONTEXT: |
| | PROJECT: | REV: | | | |
| | | | | DRAFT | |
| | | | | RE-COMMEND | |
| NOTES: 1 2 3 4 5 6 7 | | | PUBLI-CATION | | |
| 8 9 10 | | | | | |

| | | | |
|----------|--|---------------------|---------------|
| DATABASE | RECORD TYPE ID | ENTITY CLASS NAME | |
| PARTS DB | LOCATN | STOCK AREA LOCATION | |
| | | | |
| NODE: | TITLE: RECORD TYPE/ENTITY CLASS MAPPING | | NUMBER: ----- |

Figure 6-20. Record Type/Entity Class Mapping

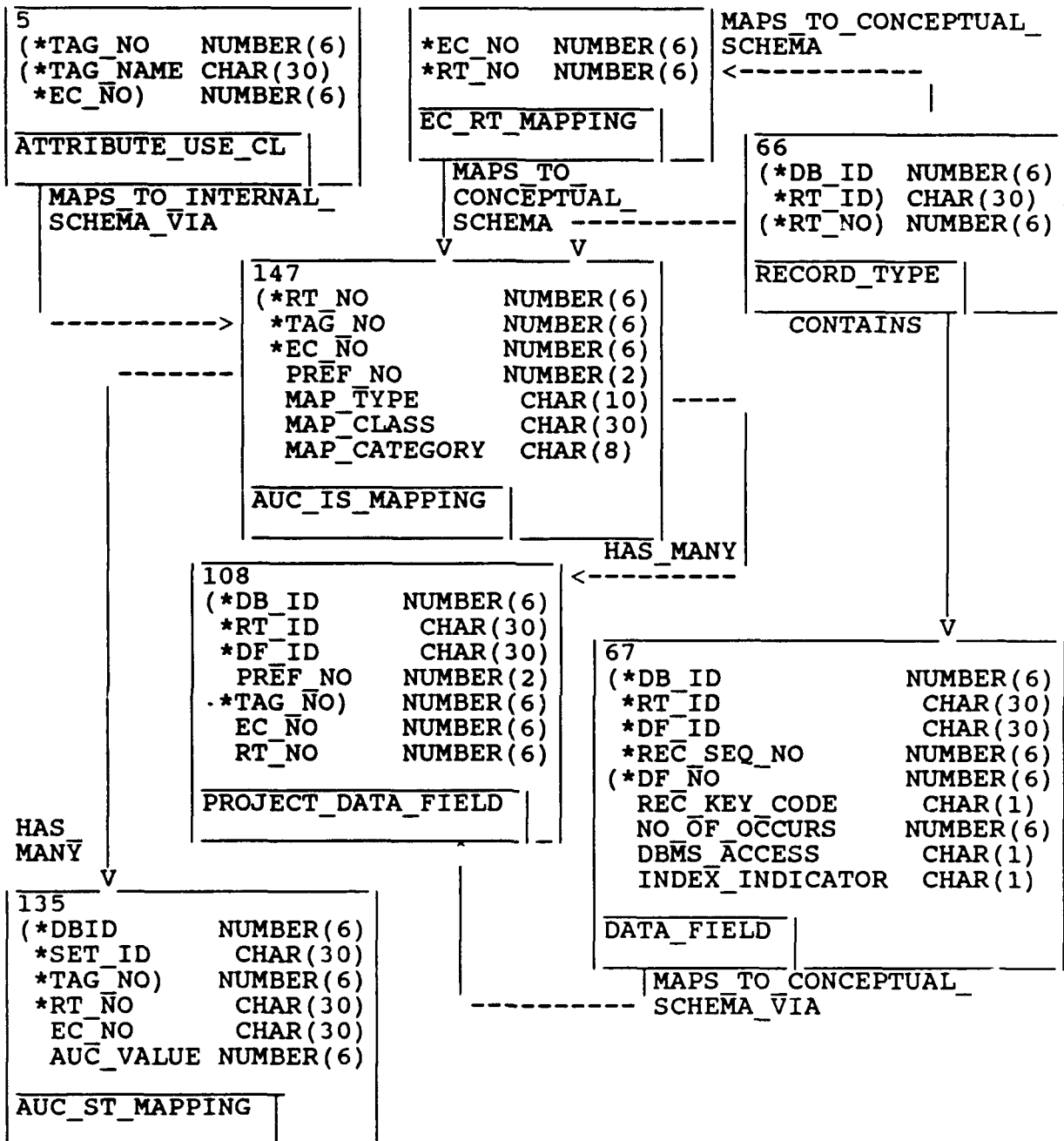


Figure 6-21. CS to IS Attribute and Relation Mapping

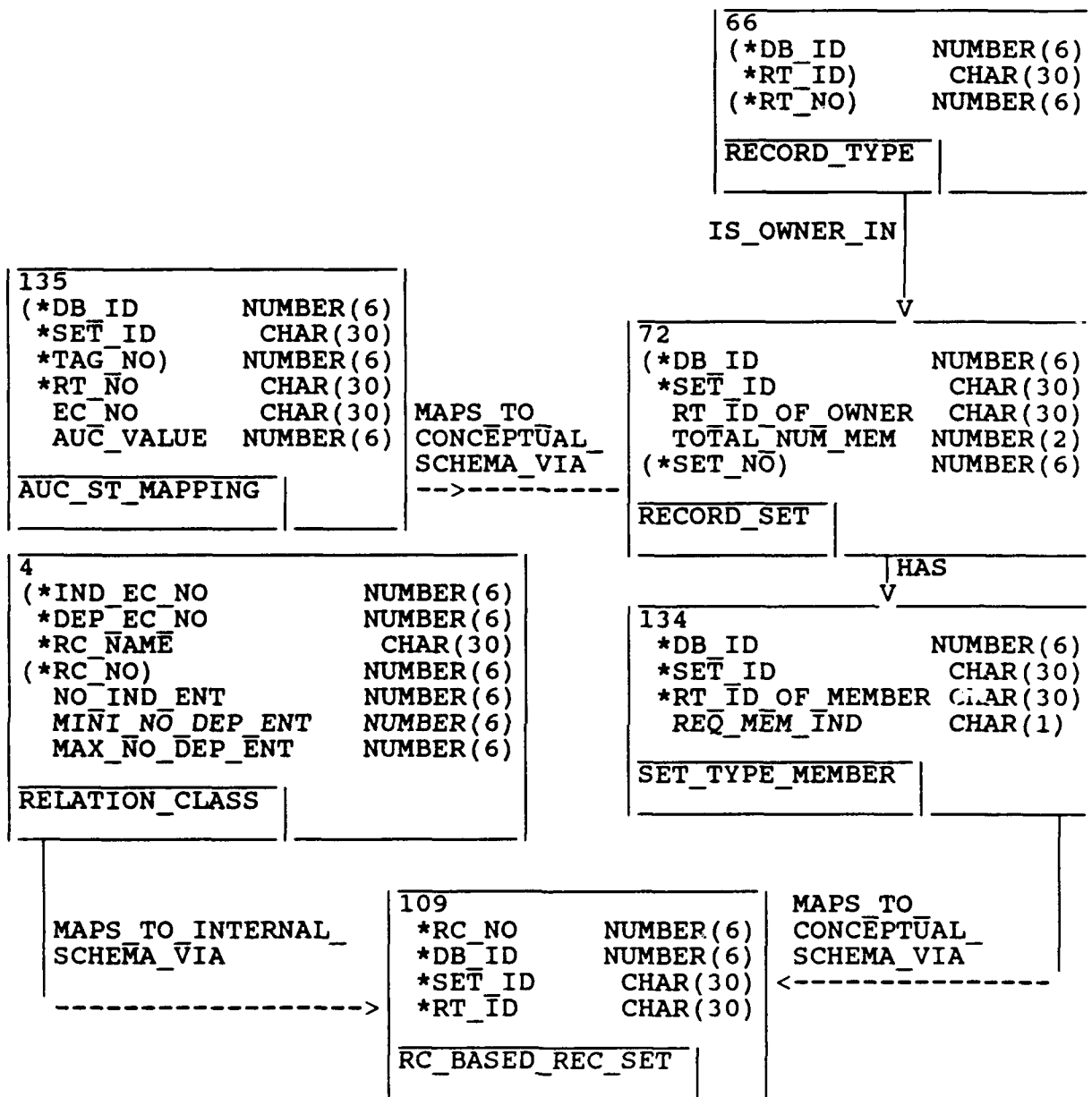


Figure 6-21. CS to IS Attribute and Relation Mapping
(Continued)

| | | | | | | |
|----------|--------------------------------|-------|--------------|-------------|----------|--|
| USED AT: | AUTHOR: | DATE: | WORKING | READER DATE | CONTEXT: | |
| | PROJECT: | REV: | | | | |
| | | | | DRAFT | | |
| | | | | RE-COMMEND | | |
| | NOTES: 1 2 3 4 5 6 7 8 9 10 | | PUBLI-CATION | | | |

| | | | | | |
|-------------------|--|--|-----------------|-------------------------|---------|
| DATABASE NAME: | | PARTS ID. | RECORD TYPE ID: | | LOCATN |
| DATA FIELD ID. | | ENTITY CLASS NAME | | ATTRIBUTE USE CLASS TAG | |
| | | | | | |
| NODE: | | TITLE: DATA FIELD/ATTRIBUTE USE CLASS MAPPING | | | NUMBER: |

Figure 6-22. Datafield/Attribute Use Class Mapping

| DB NAME | SET TYPE ID. | MEMBER RECORD TYPE ID. | IND. E.C. NAME | RELATION CLASS LABEL | DEP. E.C. NAME |
|--|------------------------|---------------------------|-------------------|-------------------------|---------------------------|
| PARTS DB | PHYSICALLY CONTROLS | LOCATN | STOCK AREA | IS COMPOSED | STOCK AREA LOCATION |
| <div> <div>NODE:</div> <div>TITLE: SET TYPE/RELATION CLASS MAPPING</div> <div>NUMBER:</div> </div> | | | | | |

Figure 6-23. Set Type/Relation Class Mapping

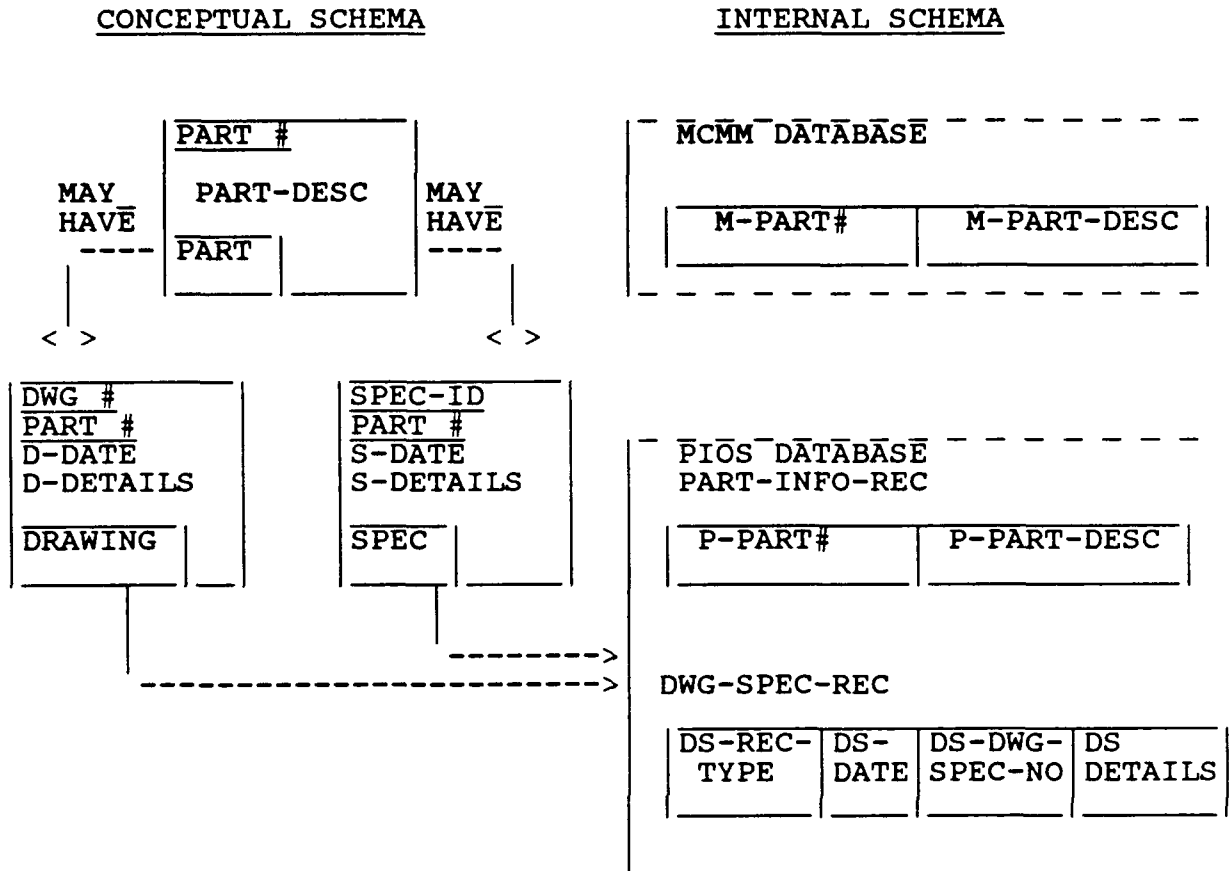


Figure 6-24. Record Union

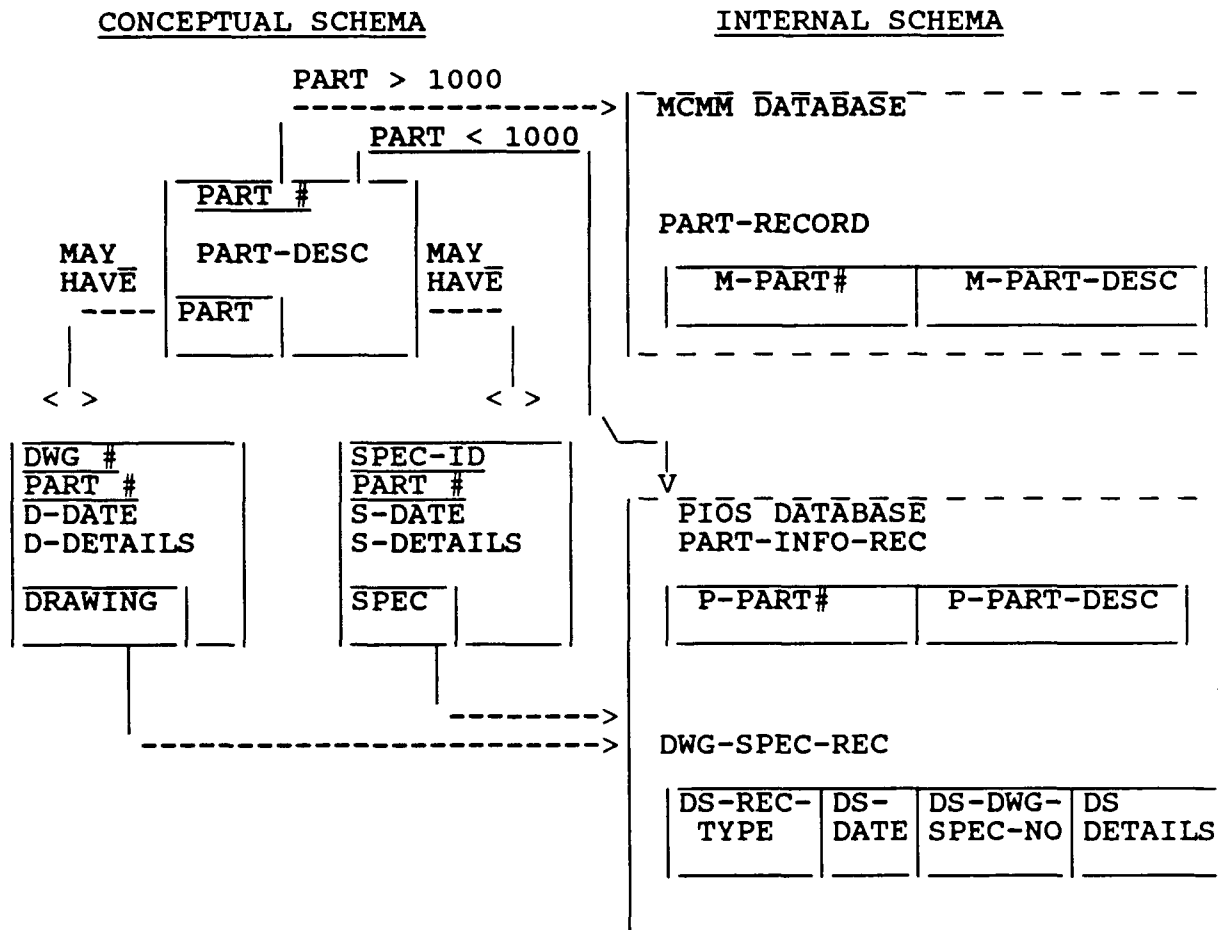


Figure 6-25. Horizontal Partition

CONCEPTUAL SCHEMA

| | |
|-----------------|--|
| CUST_NAME | |
| BILL_TO_COUNTRY | |
| BILL_TO_ZIP | |
| CUSTOMER | |

INTERNAL SCHEMA

| | |
|---------------------|--|
| CUST_NAME | |
| BILL_TO_COUNTRY_ZIP | |
| | |
| | |

Figure 6-26. CS to IS Complex Mapping Algorithms

F4, F10A

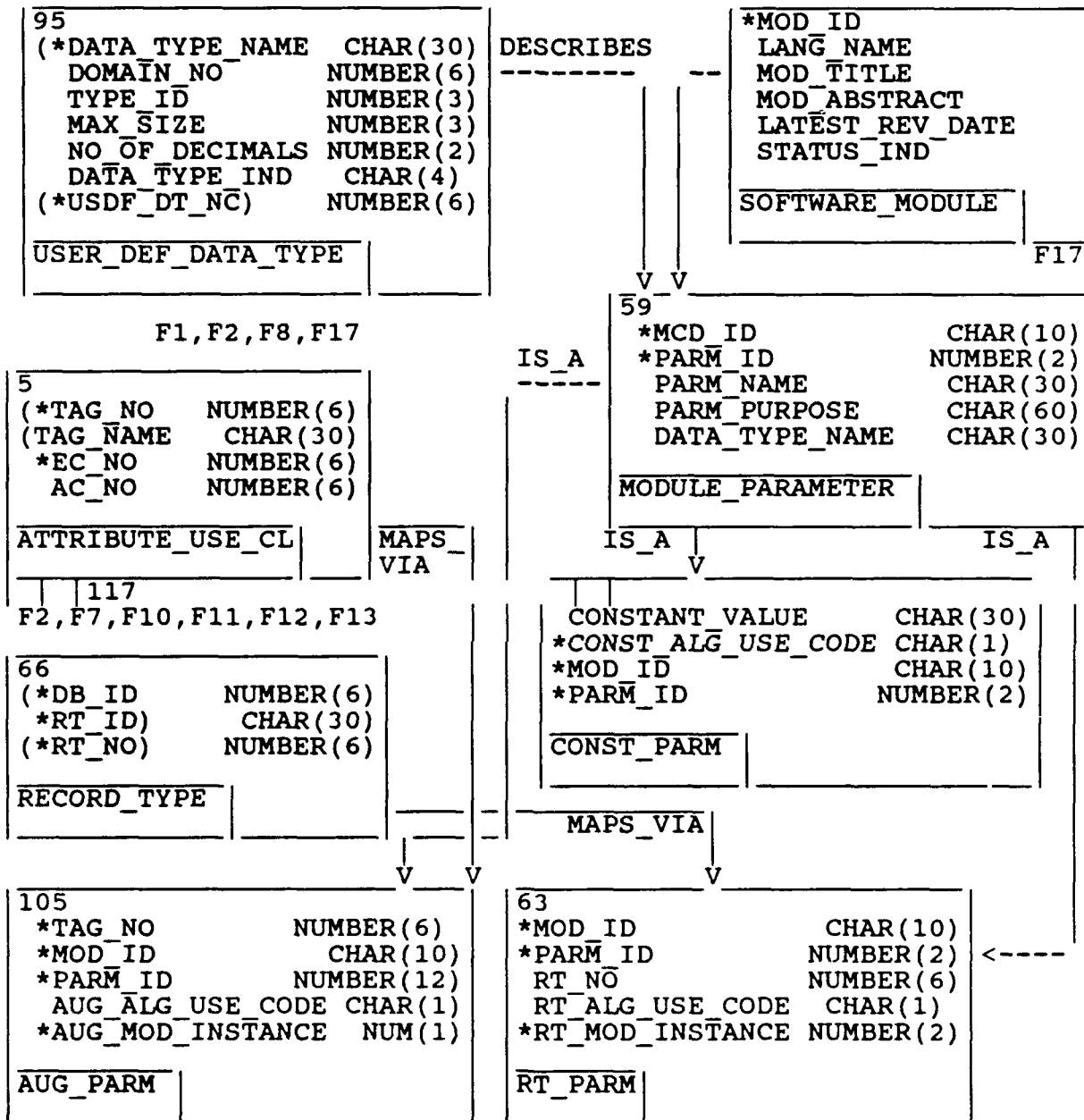


Figure 6-27. Complex Mapping Algorithm

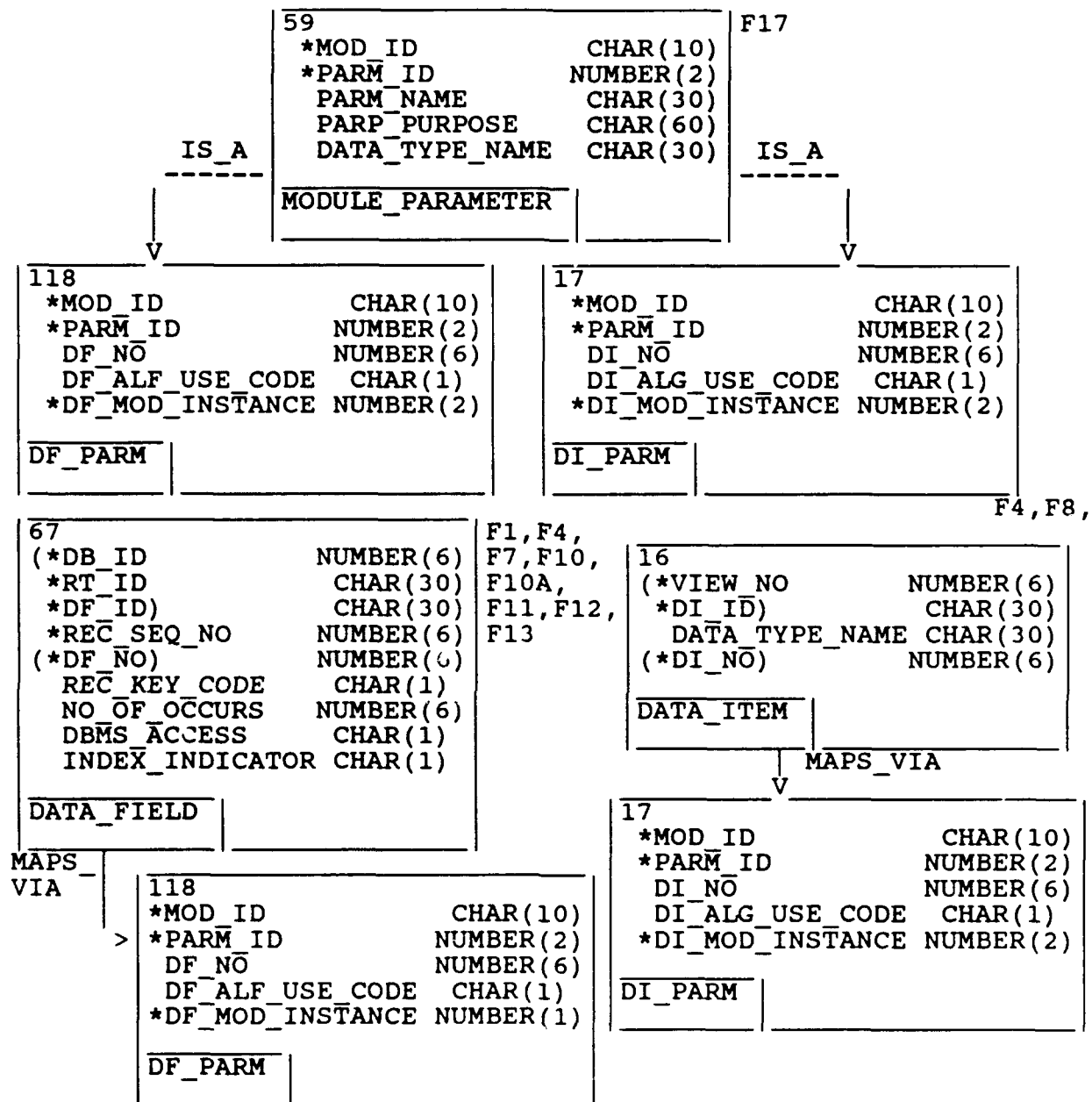


Figure 6-27. Complex Mapping Algorithm (Continued)

6.4 Modifying/Deleting IS Objects

Prior to modifying or deleting elements of the internal schema, the CDM Administrator must assess the impact of the proposed change on the other schema objects of the CDM. As stated before, whenever changes are to be made to the CDM, a CDM Impact Analysis should be run to generate reports giving information necessary to determine what other CDM objects are affected. Refer to the CDM Impact Analysis User's Manual for detailed instructions on how to use the Impact Analysis Tool.

The objective of this section is to provide the CDM Administrator with the information necessary to make these changes, determine the prerequisites before changing or dropping IS objects and to detail the options available for each IS object modification. As a general rule, the NDDL processor does not allow IS objects to be dropped if mappings exist to the conceptual schema.

The NDDL command to modify and delete the internal schema objects (i.e., data types, databases, record types, datafields and sets) are:

```
ALTER DOMAIN
ALTER DATABASE
DROP DATABASE
ALTER RECORD
DROP RECORD
ALTER FIELD
DROP FIELD
DROP SET
```

With the procurement of new hardware and DBMS software or the obsolescence of the same, the CDM Administrator might find the need to change or delete the DBMS and host definitions in the IISS environment to the CDM. The NDDL commands to accomplish this are:

```
ALTER DBMS
DROP DBMS
ALTER HOST
DROP HOST
```

The NDDL commands that change or delete CS-IS mapping definitions are:

```
ALTER MAP
DROP MAP
ALTER PARTITION
DROP PARTITION
ALTER UNION
DROP UNION
DROP ALGORITHM
DROP MODULE
```

6.4.1 Distributed Database Environment Changes

The CDM Administrator can modify a DBMS in the distributed database environment with the NDDL ALTER DBMS command. The

changes that can be made with this command are:

- * Add hosts associated with a particular DBMS

Use the ADD HOST clause. The DBMS_ON_HOST CDM Table will be updated.

- * Drop hosts associated with a particular DBMS

Use the DROP HOST clause. The entry for that DBMS and host will be deleted in the CDM DBMS_ON_HOST command. Any databases that reside on the host being dropped and utilizing the DBMS named must be dropped before this host association can be dropped. Use the NDDL DROP DATABASE command. Refer to Subsection 6.4.3 for DROP DATABASE Pre-Requisites.

- * Change the DBMS' model type

Use the MODEL clause. The DB_MODEL column in the IISS_DBMS CDM Table is updated.

DBMSs are dropped from the distributed database environment definition with the NDDL DROP DBMS command. Before the DBMS can be dropped:

- * Drop any databases operated by the DBMS. Use the NDDL DROP DATABASE command. Refer to Subsection 6.4.3 for DROP DATABASE Pre-Requisites.

- * Drop any associations with hosts.

Use the NDDL ALTER HOST command and the DROP DBMS clause. The entry for the DBMS and host is deleted from the DBMS_ON_HOST CDM Table.

The DROP DBMS command will delete the entry in the IISS_DBMS CDM Table.

The CDM Administrator can modify a host in the distributed database environment with the NDDL ALTER HOST command. The changes that can be made with this command are:

- * Add DBMSs associated with a particular HOST.

Use the ADD DBMS clause. The DBMS_ON_HOST CDM Table will be updated.

- * Drop DBMSs associated with a particular HOST.

Use the DROP DBMS clause. The entry for that host and DBMS will be deleted in the CDM DBMS_ON_HOST command. Any databases that utilize the DBMS being dropped and reside on the host named must be dropped before this host association can be dropped. Use the NDDL DROP DATABASE command. Refer to Section 6.4.3 for DROP DATABASE Pre-Requisites.

Hosts are dropped from the distributed database environment definition with the NDDL DROP HOST command. Before the host can be dropped:

- * Drop any databases that reside on the host.

Use the NDDL DROP DATABASE command. Refer to Subsection 6.4.3 for DROP DATABASE Pre-Requisites.

- * Drop any associations with DBMSs.

Use the NDDL ALTER DBMS command and the DROP HOST clause. The entry for the DBMS and host is deleted from the DBMS_ON_HOST CDM Table.

The DROP HOST command will delete the entry in the IISS_HOST CDM Table.

6.4.2 Modifying User-Defined data types

The user-defined data type allows the CDM Administrator to define to the CDM the data storage representation of a datafield. This alleviates the restriction of using only the formats allowed for the standard data type. If the format representation of the datafield changes, the data type originally specified for the datafield can be changed to another data type. This is accomplished by dropping the datafield and re-creating it. Another option available to the CDM Administrator is to change the actual format for the original data type that was specified for the datafield. This is accomplished with the ALTER DOMAIN NDDL command. The ALTER TYPE clause allows the data type to be changed to another legal type with a new size and decimal specification. The USER_DEF_DATA_TYPE CDM Table is modified with the new information provided on the ALTER TYPE clause.

Any application programs that search or update the datafields and generate application programs must be re-precompiled. These software modules will be specified on the CDM Impact Analysis reports.

6.4.3 Database Changes/Deletes

The CDM Administrator can modify the internal schema defining a database with the NDDL ALTER DATABASE command. The changes that can be made with this command are:

- * Change the database password.

Use the WITH PASSWORD clause. This clause is applicable only to ORACLE databases. The CDM Table DB_PASSWORD is updated with the new DB_PASSWORD.

- * Change the host where the database resides.

Use the TO HOST clause. The CDM Table DATA_BASE is updated with the new HOST_ID.

- * Change the schema and subschema names for CODASYL databases.

Use the SCHEMA clause. The CDM Table SCHEMA_NAMES is updated with the new names.

- * Change the VAX directory where the VAX-11 CODASYL database is stored.

Use the LOCATED AT clause. The CDM Table SCHEMA_NAMES is updated with the new DB_LOCATION.

- * Change how null characters are stored.

Use the STORES CHARACTER clause. The CDM DATA_BASE Table is updated with the new null character information.

- * Change how null integers are stored.

Use the STORES INTEGER clause. The CDM DATA_BASE Table is updated with the new null integer information.

- * Change the NTM directory specified for the database.

Use the NTM DIRECTORY clause. The CDM DATA_BASE Table is updated with the new directory prefix.

- * Add and/or drop areas associated with a CODASYL database.

Use the ADD AREAS and/or DROP AREAS clause. The CDM DATA_BASE_AREA Table is updated or deleted with the specified area names.

Databases can also be modified by adding and/or dropping record types and datafields to an existing database. Before a database can be modified in this fashion:

- * The database must be established as current for the NDDL session.

Use the NDDL ALTER DATABASE command. All subsequent NDDL internal schema commands will be applied to the current database.

Databases are deleted from the CDM by the NDDL DROP DATABASE command. Before the database can be dropped:

- Drop all software modules that either update or retrieve data from this database. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use either records or datafields from this database as input or output parameters. Use the NDDL DROP ALGORITHM command.
- Drop all the CS-IS mappings, including partition and unions that map to records and datafields in the database. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION commands.

The DROP DATABASE command will drop all entries for the database from the DATA BASE, RECORD TYPE, DATA FIELD CDM Tables. It will also drop entries from the DATA BASE AREA, DB AREA ASSIGNMENT, SCHEMA NAMES, RECORD SET, SET TYPE MEMBER and DB PASSWORD CDM Tables if any were found for the database being dropped.

6.4.4 Record Type Changes/Deletes

The CDM Administrator can modify record types already defined for a database with the NDDL ALTER RECORD command. The record type cannot be changed to another database. In order to accomplish this, the record type must be dropped and re-added. The changes that can be made to a record type with the ALTER RECORD command area:

- * Add and/or drop areas in which the CODASYL record is stored.

Use the ADD AREAS and/or DROP AREAS clause. Entries for the record are either added or deleted in the DB AREA ASSIGNMENT CDM Table.

- * Add fields to the record type.

Use the ADD FIELDS clause. The CDM DATA FIELD Table is updated with the new field and its definition.

- * Drop fields from the record type.

Use the DROP FIELDS clause. All subcomponent fields as well as entries for the field specified are dropped from the CDM DATA FIELD Table. Refer to Subsection 6.4.5 for DROP FIELD Pre-Requisites.

Record Types are deleted from the CDM by the NDDL DROP RECORD command. Before the record type can be dropped:

- Drop all software modules that either update or retrieve data from this record type. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use either the record type or its datafields as input or output parameters. Use the NDDL DROP ALGORITHM command.
- Drop all the CS-IS mappings, including partition and unions that map to the record type or its datafields. use the NDDL DROP MAP, DROP PARTITION and DROP UNION commands.

The DROP RECORD command will drop all entries for the record type from the CDM RECORD_TYPE Table. It also will delete any associated entries in the DB AREA ASSIGNMENT, RECORD SET, SET_TYPE_MEMBER and DATA_FIELD CDM Tables.

The DROP SET command will drop all entries for the set type from the RECORD SET and SET TYPE MEMBER CDM Tables. Set types and their mappings are applicable to CODASYL databases. The set type will not be dropped; however, if the set maps to a tag or a relation class (see Subsection 6.3.1, "Loading CS to IS Mappings"). If a mapping exists, it must be dropped using the NDDL DROP MAP command.

6.4.5 Datafield Changes/Deletes

The CDM Administrator has two options when changing a datafield. It can be dropped and re-added from a record type by using the ADD FIELD and DROP FIELD clause of the ALTER RECORD command, explained in Subsection 6.4.4. Or some of the fields characteristics can be modified with the ALTER FIELD NDDL command. The datafield cannot be changed to another record type or database. The changes that can be made with this command are:

- * Change the data type associated with the datafield or eliminate the data type association on a CODASYL group field.

Use the DATA TYPE clause. The CDM DATA_FIELD Table is updated.
- * Change, add or remove the depending on field.

Use the DEPENDING ON clause. The CDM DATA_FIELD Table is updated.
- * Change, add or remove the redefines datafield name.

Use the REDEFINES clause. The CDM DATA_FIELD Table is updated.
- * Change the datafield's "known" status by the DBMS.

Use either the word KNOWN or UNKNOWN. The CDM DATA_FIELD Table is updated.
- * Change the datafield's status as key or non-key.

Use the UNIQUE KEY, DUPLICATE KEY or NOT KEY clause, whichever is desired. The CDM DATA_FIELD Table is updated.

Datafields are deleted from the CDM by the NDDL DROP FIELD command. Before a datafield can be dropped:

- Drop all software modules that either update or retrieve data from this datafield. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use this datafield as an input or output parameter. Use the NDDL DROP ALGORITHM command.

- Drop all the CS-IS mappings, including partition and unions that map to the datafield. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION commands.

6.4.6 Modifying/Deleting CS-IS Mappings

The CDM Administrator can change the mapping definitions of the internal schema objects to their appropriate conceptual schema objects at three different levels. All these changes are accomplished by different variations of the NDDL ALTER MAP command. The changes that can be made to entity class to record type mappings are:

- * Change the distributed update rule. (i.e., update all copies or only preferred copy.)

Use either ALLOW UPDATE or DISALLOW UPDATE clause. The DISTRIBUTED_RULES CDM Table is updated accordingly.

- * Change the distributed retrieval rule. (i.e., select primary copy or preferred copy on host.)

Use either ALLOW RETRIEVAL or DISALLOW RETRIEVAL clause. The DISTRIBUTED_RULES CDM Table is updated accordingly.

- * Add records that map to the entity to allow for horizontal partition mappings and/or duplicated copies.

Use the ADD RECORD clause. The EC_RT_MAPPING CDM Table is updated with the new entity to record mapping definition.

- * Drop record that map to the entity.

Use the DROP RECORD clause. The entry for the entity and record is deleted from the EC_RT_MAPPING CDM Table. Before the entity to record mapping can be dropped:

- Drop all software modules that either update or retrieve data from this record type. Use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use either the record type or its datafields as input or output parameters. Use the NDDL DROP ALGORITHM command.
- Drop all the CS-IS mappings, including partition and unions that map to the record type's datafields. Use the NDDL DROP MAP, DROP PARTITION and DROP UNION commands.

Changes to the mapping definitions at the Attribute Use Class to Datafield level are also accomplished with the ALTER MAP command. The changes available at this level are:

- * Change the preference number of the mapping.

Use the TO PREFERENCE clause. The AUC_IS_MAPPING CDM Table is updated. The preference will be switched if both preferences specified exist.

- * Change the map category.

Use either the word ACTIVE or PASSIVE. The AUC_IS_MAPPING CDM Table's MAP_CATEGORY is changed. Map categories are associated with preference numbers. Preferences 1 through 49 are ACTIVE, while 50-99 are PASSIVE.

- * Change the mapping classification.

Select the desired mapping classification. The AUC_IS_MAPPING CDM Table's MAP_CLASS is changed. This clause is mainly for descriptive purposes.

- * Additional datafields may be mapped to the attribute use class to support additional horizontal partition fragments for a stated preference.

Use the ADD FIELD clause. New entries are added to the AUC_IS_MAPPING CDM Table for the mapping definition. A mapping must exist from the attribute's entity to the datafield's record type before this mapping can be added.

- * Attribute use class to datafield mapping definitions for a stated preference can be dropped when a horizontal partition fragment is no longer needed.

Use the DROP FIELD clause. The AUC_IS_MAPPING CDM Table entry is deleted. If this is the last mapping for the attribute's entity, the entity to record mapping definition is deleted from the EC_RT_MAPPING CDM Table. Before this mapping definition can be dropped:

- Drop all software modules that either update or retrieve data from this datafield. use the NDDL DROP MODULE command.
- Drop all complex mapping algorithms that use the datafield as input or output parameters. Use the NDDL DROP ALGORITHM command.

- * Set value mappings can be added or dropped.

Use the ADD SET or DROP SET clause. Mapping definitions are either added or deleted from the AUC_IS_MAPPING and AUC_ST_MAPPING CDM Tables.

- * Change the values for existing attribute use class to set mappings.

Use the ALTER SET clause. The AUC_VALUE in the AUC_ST_MAPPING CDM Table is updated with the specified value.

CODASYL databases make use of relation class to record relationship (i.e., set type) mappings. These mapping definitions are also changed with the NDDL ALTER MAP command. The changes to these mappings are:

- * Add a new set type of a multi-member set to be mapped to the relation class.

Use the ADD SET clause after specifying the relation class name. The RC_BASED_REC_SET CDM Table is updated with the new mapping definition provided no other set maps to the relation.

- * Drop a set type mapped to the relation class.

Use the DROP SET clause. The entry in the RC_BASED_REC_SET CDM Table is deleted for the relation class to the specified set type mapping.

CS to IS mapping definitions at all levels are deleted from the CDM with the NDDL DROP MAP command. A mapping can be dropped for all attribute use classes of an entity or for a specified attribute use for a stated preference mapping. When more than one preferred mapping exists for an attribute use, dropping the first preference is not permitted. The secondary preference must be dropped first. The CDM Tables that are populated at the different mapping levels are deleted from when the DROP MAP command is issued. Individual Relation class to record set mappings are dropped with the ALTER MAP command, as described above. No mapping definition for an attribute use class will be dropped if its used as an input or output parameter in a complex mapping algorithm. Use the NDDL DROP ALGORITHM command first.

6.4.7 Record Union Changes/Deletes

The CDM Administrator can modify existing record union definitions in the CDM with the NDDL ALTER UNION command. The changes that can be made to the record union are:

- * Add entities to be unioned in the record type.

Use the ADD ENTITY clause. Another entry is added to the ECRTUD CDM Table.

- * Drop entities that were included in the record union.

Use the DROP ENTITY clause. An entry is deleted in the ECRTUD CDM Table for the record union specified.

- * Change the union discriminator or add and delete conditions to the union discriminator.

First, use the DROP ENTITY clause to drop the entity and its union discriminator from the record union. Then re-add the entity to the record union with the ADD ENTITY clause specifying the changed union discriminator and its new condition(s).

Record unions are deleted from the CDM by the NDDL DROP UNION command. This command will drop all entries from the ECRTUD CDM Table for the specified record type. The DROP UNION command will not effect the CDM definition of the record type, entity or any of its mappings.

6.4.8 Horizontal Partition Changes/Deletes

The CDM Administrator can modify existing horizontal partition definitions in the CDM with the NDDL ALTER PARTITION command. The changes that can be made to the horizontal partition are:

- * Add record type partitions for the entity.

Use the ADD RECORD clause. Another entry is added to the HORIZONTAL_PART CDM Table.

- * Drop record types that were partitions of the entity.
Use the DROP RECORD clause. An entry is deleted in the HORIZONTAL_PART CDM Table for the record partition specified.

A horizontal partition, by its definition, must consist of at least two record types. After the adding and dropping of record types (i.e., altering the horizontal partition definition) the entity's horizontal partition must consist of at least two record types.

Horizontal partitions are deleted from the CDM with the NDDL DROP PARTITION command. This command drops record partition entries in the HORIZONTAL_PART CDM Table for the entity names. Before a horizontal partition is dropped, the CDM Administrator must drop any partition mappings of the specified entity.

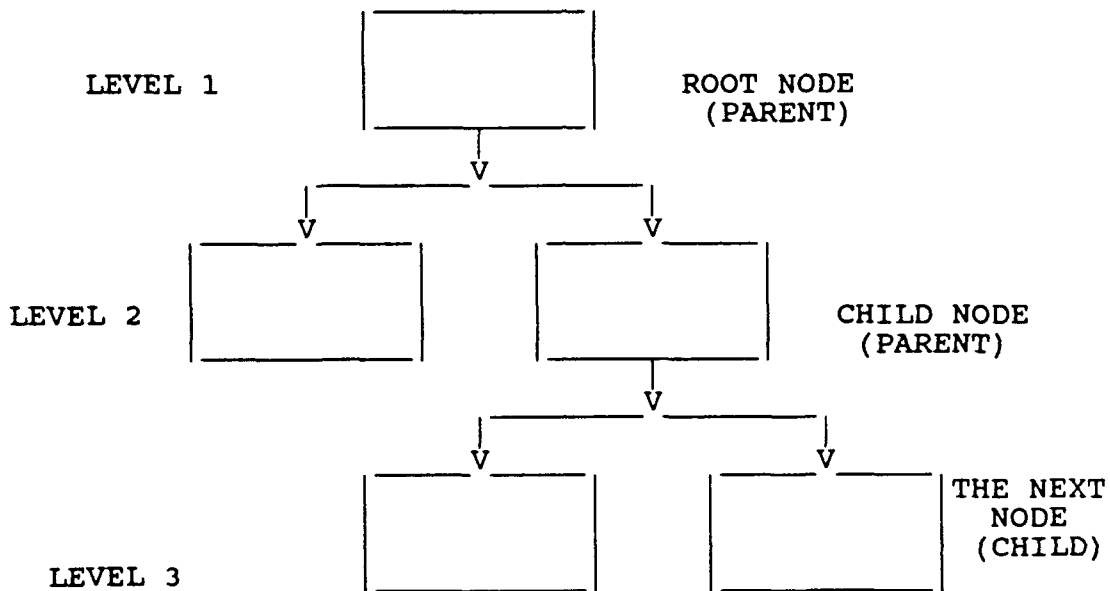
6.5 Specific Considerations

The CDM supports the definition of a number of database models and file structures that are not fully implemented by the precompiler. These database models and file structures are described in the following subsections. An explanation is provided on the clauses of the NDDL commands that are pertinent only to these database models and their specific CS-IS mapping methodologies.

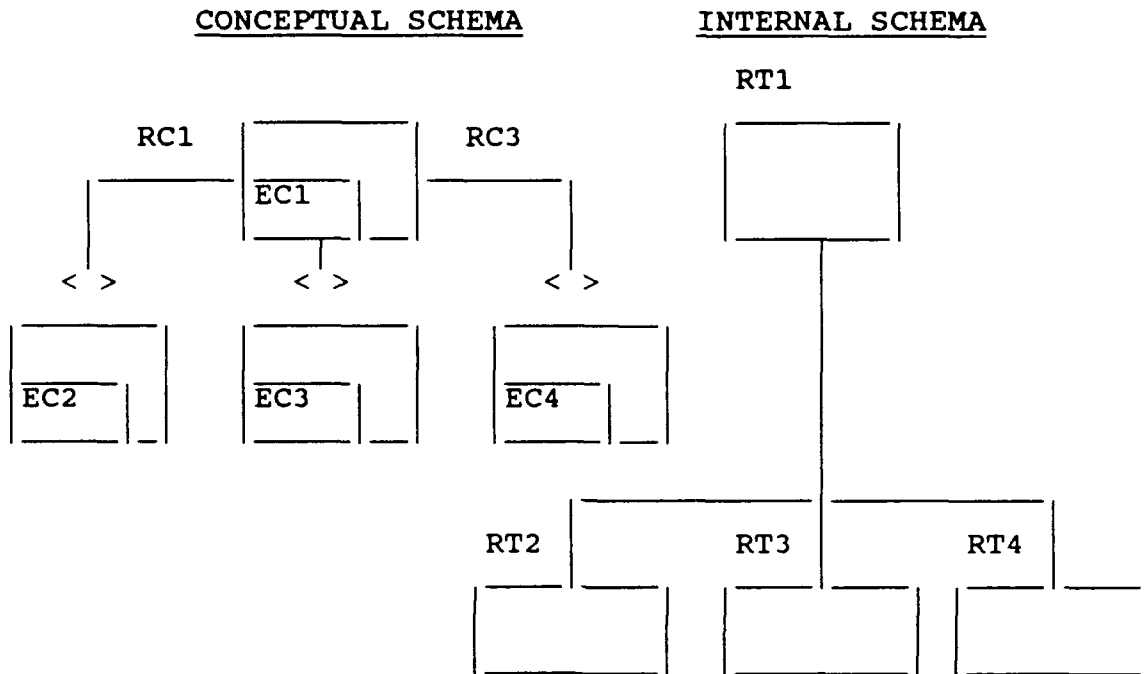
6.5.1 IMS Specific Considerations

Whereas the basic construct of the CODASYL model is a set and complicated structures can be built from sets, the IMS model represents data in the form of a tree structure. A tree consists of different levels of entities referred to as nodes. A node can have many occurrences, that is, sets of data values for its data items. Each higher level implies dominance over the levels below it, thus creating a hierarchy. The highest level contains only one node called a root node. All nodes, with the exception of the root, must be connected at a level above it. The node at the higher level is called a parent node and "owns" all of the lower level nodes in the limb. The node at the lower level is called a child node. A child node must have one and only one parent node.

A parent node can have none, one, or many nodes connected to them as children. There can be many occurrences of a specific child node under a single parent. A parent and its children at each level are considered a physical tree. A database may consist of many of these physical trees. Even though the set construct is not supported by the IMS model, a parent and all of its children are to be considered analogous to a set. The following example depicts the IMS hierarchical model.



A node is called an IMS segment. A logical record in the database consists of a root and all of its children. A database record can consist of a tree with up to 15 levels. In essence, many logical relations (relation classes) could be combined into one physical hierarchy. This is called a regular hierarchy and is defined via an IMS Database Definition (DBD).



The CS-IS mapping for a regular hierarchy involves the following:

- o Each parent-child relationship within the hierarchy maps to a relation classes.
- o The parent in each relationship maps to the entity class that is independent in that relation class.
- o The child in each relationship maps to the entity class is dependent in that relation class.

In the preceding example:

RT1 maps to EC1
RT2 maps to EC2
RT3 maps to EC3
RT4 maps to EC4
RT1:RT2 maps to RC1
RT1:RT3 maps to RC2
RT1:RT4 maps to RC3

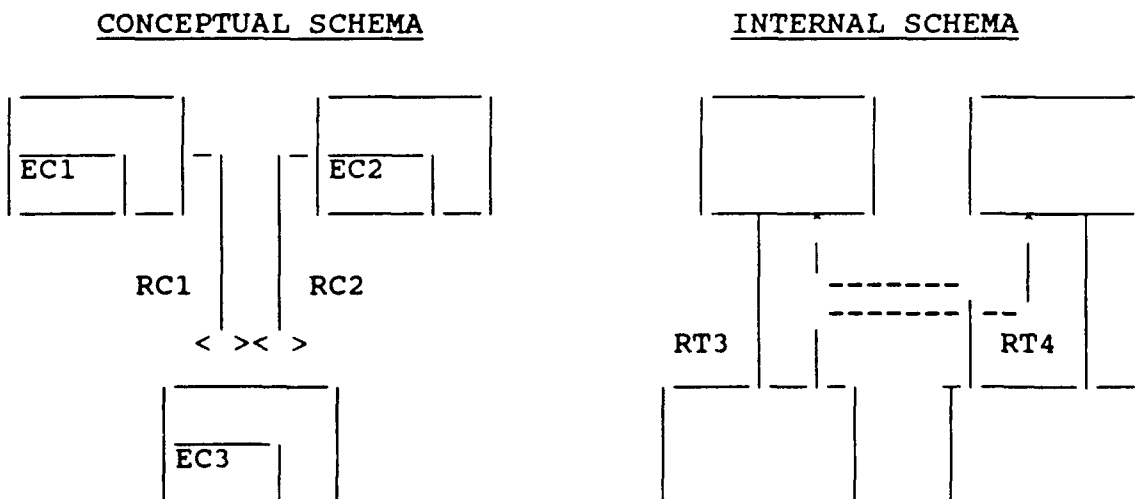
The previous diagram illustrates that each dependent segment has a parent segment and exists as one element in a child-parent relationship. These relationships can have both a physical and a logical form. The physical form of the parent-child relationship is a consequence of (1) the definition of a given data base and

(2) the method by which the data elements are stored. The logical relationship is established solely by express definition and exists externally to any physical organization constraints.

IMS has an additional relationship called a "twin". As with the parent-child relationship, two forms of twins exist: physical and logical twins. Physical twin segments are multiple occurrences of a common segment format. At the root segment level, the set of physical twins is the set of all root segment occurrences of a given database. At the dependent segment level, a set of physical twins is the set of physical child occurrences for a given segment format within a hierarchy. At the logical level, twins are multiple occurrences of a common segment format having a common logical parent. The physical and logical concepts give IMS the capability of storing network type relationships (sets) between entities. These network physical structures are viewed by users and programmers as one or more hierarchical views.

Every logical relationship involves the use of three segments; no more, no less. Two of these segment types (the physical parent and the logical parent) can exist in separate databases or they may exist in the same physical database. The third segment type (the logical child) is used to construct the logical linkage. The three segments involved in a logical relationship are an instance of a non-specific many-to-many membership set type.

A nonspecific membership set type whose cardinality is many-to-many, is one in which (1) each member of entity class "1" is related to zero, one, or many members of entity class "2" and (2) each member of entity class "2" is related to zero, one, or many members of entity class "1." Such a relation class is refined, as shown in Figure 4-7, before it is incorporated into the conceptual schema.



The CS-IS mapping for a many-to-many membership set type involves the following.

- o Each parent segment has a primary mapping to one entity class.
- o The child segments have a primary mapping to a single entity class (RT4 may or may not exist physically on the database depending on the IMS options that were chosen).
- o Two IMS DBDs are required and map to one-to-many relation classes.
- o The entity classes to which the parents map are independent in their respective relation classes.
- o The entity class to which the children map is dependent in that relation class.

In the preceding example:

```
RT1 maps to EC1
RT2 maps to EC2
RT3 maps to EC3
RT4 maps to EC3
RT1:RT3 maps to RC1
RT2:RT4 maps to RC2
```

6.5.1.1 IMS NDDL Specifics

This section deals with the differences in the CDM, specific to IMS databases, and the NDDL commands that populate these tables or columns. The CDM Tables pertinent to an IMS database definition are in Figure 6-28. Two tables are specific to IMS databases, IISS_PSB and PSB_PCB. These tables hold program status block information for the database. A program status block is used in IMS to group application views of databases into a single runtime unit. Actually, it is an area of memory used by the IMS DBMS in communicating with application processes.

The IISS PSB CDM Table is populated and modified by the DEFINE PSB, ALTER PSB and DROP PSB NDDL commands. These commands specify which host computer in the distributed database environment the program status block resides.

The PSB_PCB CDM Table is populated by the DEFINE PCB NDDL command. The PSB_PCB Table maintains the cross reference of the Program Communication Blocks (i.e., PCBs) and the Program Status Block (i.e., PSB). A PCB defines the hierarchical structure or sub-structure of a logical or physical database that can be viewed by an application. The PSB, as mentioned before, maintains the runtime link to the DBMS for an application. This command is identical to the DEFINE DATABASE command, but the keyword PCB must be used for an IMS database.

The WITH POSITION clause on this command is pertinent only to an IMS database. This clause updates the feedback length in the PSB_PCB CDM Table. The key feedback length is used in the Program Communication Block to define the maximum size of the concatenated keys from each segment in the hierarchy, starting at

the root and ending at the bottom child segment. Since there may be many terminating child segments, key feedback length must be set to its maximum.

Other NDDL commands vary only in terminology. Instead of using the word "set", IMS calls the relationships between parent and child records "paths". The IMS DBMS also calls record types segment types and datafields data elements.

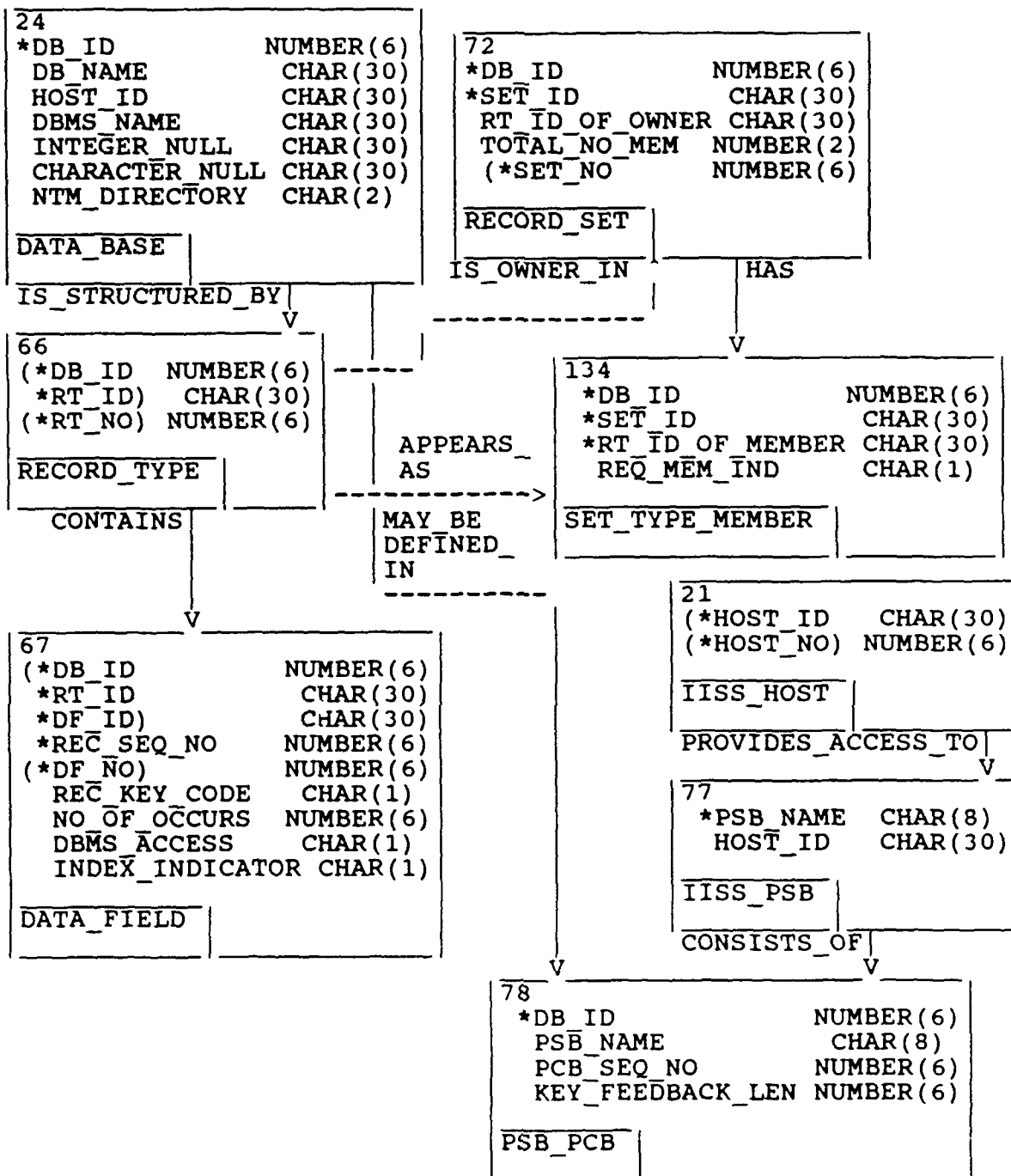


Figure 6-28. IMS Internal Schema

6.5.1.2 IMS CS to IS Mapping Methodology

All the modeling forms are pertinent when mapping IMS databases to the conceptual schema. First, the CDM Administrator determines the primary mapping for each segment type. Usually, it is easier to map an IMS database by starting with the root segment and working down each limb of the tree. A child member should not be mapped until its parent segment type has been mapped. The CDM Administrator determines what sort of "real-world" thing the segment type represents. Each instance of a segment type contains data about a specific person, place, object, etc., that is significant to the enterprise. With IMS, all of the instances of the same segment type are about the same sort of thing.

Next, the CDM Administrator determines which entity class in the conceptual schema represents the same sort of thing as this segment type identified prior. This primarily involves finding the entity class whose definition corresponds to the intent of the segment type. Comparing the key classes, attribute use classes, and relation classes of the entity classes to the keys, data elements and parent-child relationships of segment types can be helpful also. If the segment type represents several sorts of things, it will map to several entity classes, one for each sort of thing (see Subsection 6.1.1.4, "Unions"). If none of the entity classes represent what the segment type does, either the segment type exists only to provide a logical relationship or the conceptual schema must be expanded (see Subsection 4.3). Once the entity class to which the segment type maps is determined, the information is recorded in the Record Type/Entity Class Mapping Form.

A few segment types do not represent real-world things; they exist to provide database pointers between logically related segments. Such segment types do not map to any entity classes and can be ignored.

Next, the CDM Administrator uses the Data Field/Attribute Use Class Mapping Form to assist in determining the primary mappings for each data element. The content of the data element is analyzed to result in what sort of data about "real-world" things it represents. If the segment type that contains the data element represents more than one sort of thing, i.e., it has more than one mapping, the data element may contain several sorts of data. All of these must be identified.

A few data elements may not contain data about "real-world" things and exist for technical reasons only. Examples include segment codes and segment activity dates. Such data elements do not map to any attribute use classes and can be ignored.

The way the CDM Administrator determines which attribute use class in the conceptual schema represents the same sort of data as the data element is by finding the attribute use class whose definition or migration path corresponds to the intent of the data element. The first place to look is the entity class to which the segment type maps. If the segment type maps to

more than one entity class, the data element may map to an attribute use class in each. The value in the data element in each instance of the segment type must be the same as the one in the attribute use class in the corresponding instance of the entity class. If two or more inherited attribute use classes that come from the same owned attribute use class have identical values in every entity instance, the data element may map to some or all of them.

If none of the attribute use classes in the mapped-to entity classes correspond to the data element, the next places to look are the entity classes that are related to those entity classes. Again, the value in each segment instance must be the same as the value in the corresponding entity instance. If the attribute use class is not in any of these entity classes, the search must be widened to include the entity classes that are related to them. This continues until the proper attribute use class is found or until it is determined that a new attribute class must be added to the conceptual schema (see Subsection 4.3). This mapping information is recorded on the Data Field/Attribute Use Class Mapping Form.

The CDM Administrator's next step is to determine if any secondary mappings are needed for each data element by finding the data elements in the segment type that map to attribute use classes that are not in the entity class to which the segment type maps. This can be done by comparing the entity class names entered on the Data Field/Attribute Use Class Mapping Forms for the segment type to those that are entered on the segment type's Record Type/Entity Class Mapping Form. If an entity class name is on the first form but not on the second, then that entity class must be joined with the one to which the segment type maps.

Other entity classes might need to be identified to complete the join structure. The entity classes that must be joined to form the segment type must form one or more join structures as described in Subsection 6.1.1.3. If the join structures are not contiguous, one or more additional joins may be needed. For example, if the segment type in Figure 6-14 maps to EC4 and involves joins with EC1 and EC3, it also must have a join with EC2. Without it, EC1 cannot be joined to the EC3-EC4 join result. The join must involve EC2 even though none of its attribute use classes map to data elements in the segment type. Draw diagrams of these segment types that involve joins on the Record Type Join Structure Diagram Form.

The final step for the CDM Administrator when following this methodology is to determine the mapping for each parent-child relationship. What sort of relationship between "real-world" things the set type represents must be established. If the set type has more than one child segment type, each must be considered separately. If either the parent or the child segment type has no mapping to an entity class, the set type will have no mapping to a relation class so it can be ignored.

The CDM Administrator determines which relation class in the conceptual schema represents the same sort of relationship as the set type. Usually, this is the relation class whose

independent entity class maps to the parent segment type and whose dependent entity class maps to the child segment type. Fill out a line on a Set Type/Relation Class Mapping Form once the relation class to which the set type maps is determined.

Once this CS-IS mapping methodology has been followed and completed, the CDM Administrator loads these mappings as described in Subsection 6.3.1, "Loading CS to IS Mappings".

6.5.2 VSAM Specific Considerations

The Virtual Storage Access Method (VSAM) is a component of the IBM operating system's data management services. VSAM supports both direct and sequential processing. VSAM data sets cannot be accessed by any other access method.

VSAM support consists of the following:

- o Three data sets organizations: Entry-Sequenced Data Sets (ESDS). Key-Sequenced Data Sets (KSDS), and Relative-Record Data Set (RRDS). They are supported on DASD (Direct Access Storage Devices) only.
 - A VSAM ESDS is a sequential data set (similar to a SAM data set).
 - A VSAM KSDS is a sequential data set with an index (similar to an ISAM data set).
 - A VSAM RRDS is a data set with preformatted slots for fixed length records to be accessed by a record number (similar to a DAM data set).

NOTE: As VSAM RRDSs are rarely used, they are excluded from this document.

The mappings for an ESDS and KSDS are identical to those discussed in Subsection 6.5.3.

6.5.3 Sequential Files Specific Considerations

The mapping from the Conceptual Schema to a sequential file is very straightforward where:

- o Nonspecific relationships have been resolved.
- o Keys have been migrated.
- o No role names are used.

In mapping to a sequential file:

- o Each entity class becomes a record.
- o Each attribute of an entity becomes a data field in the corresponding record.
- o The key of each entity becomes the primary key in the corresponding record.

If relationships between sequential files are implied (foreign keys have been migrated), please refer to Subsection 6.1.2.1., "Relational Database Modeling Forms" for the CS to IS mapping methodology to follow.

6.5.3.1 Sequential File NDDL Specifics

Very few differences exist between defining sequential file internal schema objects to the CDM and defining the same for relational databases. For each sequential file, use the NDDL DEFINE DATABASE command (see Subsection 6.2.3.1, "Loading Relational Databases"). For each sequential file's record type, use the NDDL DEFINE RECORD command. The difference between sequential file records and relational database records is that sequential file records can group its datafields, i.e., relational databases only support elementary datafields. Follow the instructions outlined in Subsection 6.2.4.2, "Loading CODASYL DBMS' Record Types".

6.5.3.2 Sequential File CS to IS Mapping Methodology

Sequential files utilize the Entity Class/Record Type Mapping Form and the Data Field/Attribute Use Class Mapping Form. The CDM Administrator first determines the mapping for each record type. What sort of "real-world" thing the record type represents is established. Each instance of a record type contains data about a specific person, place, object, etc. that is significant to the enterprise. The CDM Administrator determines which entity class in the conceptual schema represents the same sort of thing as the sequential file's record by finding the entity whose definition corresponds to the intent of the record. The entity class to which the record maps is recorded on the Record Type/Entity Class Mapping Form.

Next, the CDM Administrator determines the mapping for each datafield by establishing what sort of data about "real-world" things that the datafield contains. A one-for-one mapping between attributes of an entity and datafields of its corresponding record should always exist. Finding which attribute use class in the conceptual schema represent the same sort of data as the datafield involves finding the attribute use class whose definition or migration path corresponds to the intent of the datafield. This attribute use class and datafield to which it maps is recorded on the Data Field/Attribute Use Class Mapping Form.

A few datafields might not contain data about "real-world" things; they exist for technical reasons only. Examples include record codes and record activity dates. Such datafields do not map to any attribute use classes and can be ignored.

Once this CS-IS mapping methodology has been followed and completed, the CDM Administrator loads these mappings as described in Subsection 6.3.1, "Loading CS to IS Mappings".

SECTION 7

MAINTAINING EXTERNAL SCHEMAS AND MAPPINGS

7.1 Methodology Overview

This section provides the CDM Administrator with the methodology for building and maintaining external schemas and or mapping them to the conceptual schema. The tables of the CDM database that are populated to describe external schemas and CS-ES mappings are shown in Figure 7-6. As mentioned in Section 5, the external schema objects are views and data items. A user view is equivalent to a table in a relational database; a data item to a column of that table. The format representation of data items can be defined with user defined data types. The mapping between the conceptual schema and external schema only has one level, attribute use class to data item.

This section will explain the basic concepts of user views and how to determine their mappings to the conceptual schema. Forms are used to assist in the CS-ES mappings and instructions on their use are provided. The NDDL commands necessary to initially load user views and CS-ES mappings are described along with the NDDL commands to change the external schema objects.

7.1.1 External Schemas AND CS-ES MAPPING STRUCTURE

A CS-ES mapping is intended to show which components of an external schema correspond to those of the conceptual schema. A data item maps to an attribute use class if they both are the same kind of data about real-world things. A data item is not to map to more than one attribute use class. In Figure 7-1, the EMP-NAME, DEPT-NAME, and SPOUSE-NAME data items each have only one attribute use class to which to map. If there is more than one attribute use class to which a data item could map, the choice depends on which entity class each attribute use class is in. There must be one entity class that has one entity instance for each row in the user view. If one of the attribute use classes is in that entity class, the data item maps to it. In Figure 7-1, the Employee entity class has one instance for each row in the EMP-MAST user view, so that EMP-NO and DEPT-NO data items map to the equivalent attribute use classes are that entity class. If none of the attribute use classes are in that entity class, the data item maps to the one in the entity class that is most closely related to that entity class. Thus, DIV-NO maps to Div No in the Dept entity class because that entity class is closer to Employee than the Division entity class.

The following subsections (7.1.1.1 - 7.1.1.2) present two subjects to consider when dealing with CS-ES mappings. They are not mutually exclusive; a user view can involve neither, either, or both.

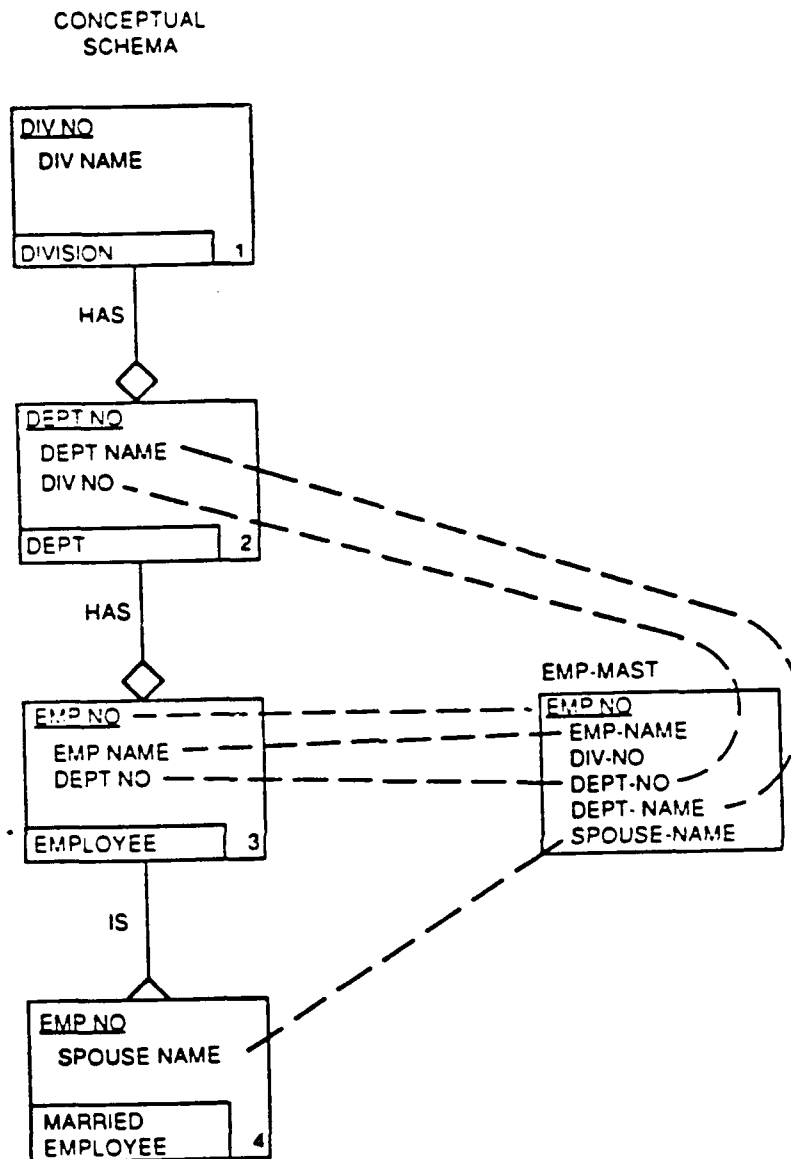


Figure 7-1. Data Item/Attribute Use Class Mappings

7.1.1.1 Vertical Partitions

An entity class is vertically partitioned when some of its attribute use classes map to data items in one user view and others map to those in another. An entity class can have several vertical partitions.

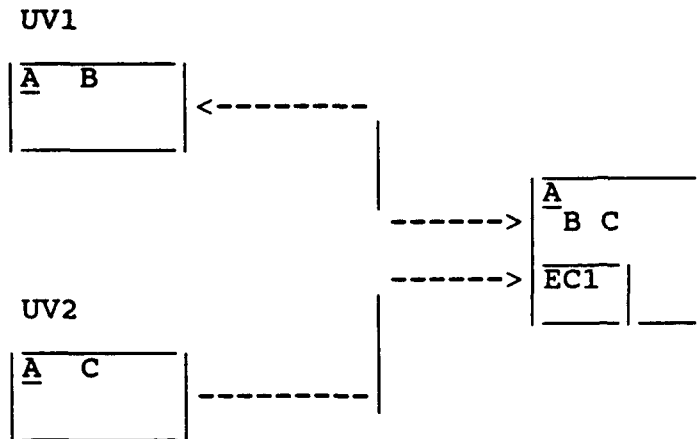


Figure 7.2. Vertical Partition

7.1.1.2 Joins

If the data items in a user view map to attribute use classes in two entity classes, those entity classes must be combined to form that user view. This is done with a relational "join" operation, which concatenates the entity instances of one entity class with those of the other. These two entity classes must be directly related by a relation class so that their entity instances can be matched using the key class of the independent and the corresponding inherited attribute use class(es) of the dependent.

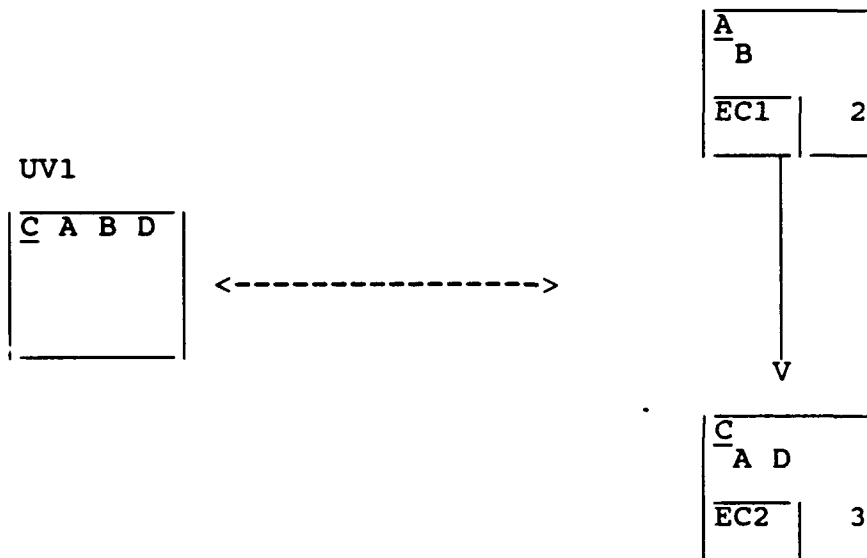
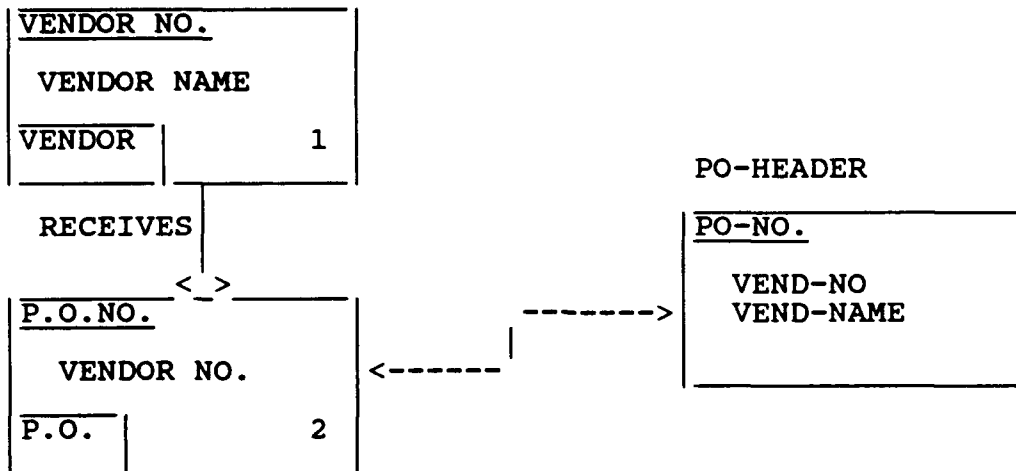


Figure 7.3. Entity Joins

If the relation class cardinality is one-to-many, each independent entity instance is concatenated with each entity instance that is dependent on it. In the first example in Figure 7-4, each PO-HEADER instance is formed by concatenating a Vendor instance with a PO instance based on identical values in Vendor No. If a Vendor instance has no dependent PO instances, it is not represented by a PO-HEADER instance. This produces one row in the use view for each instance in the dependent entity class. Since a relational join cannot form user view rows with repeating data items, this concentration cannot be done from dependent to independent.

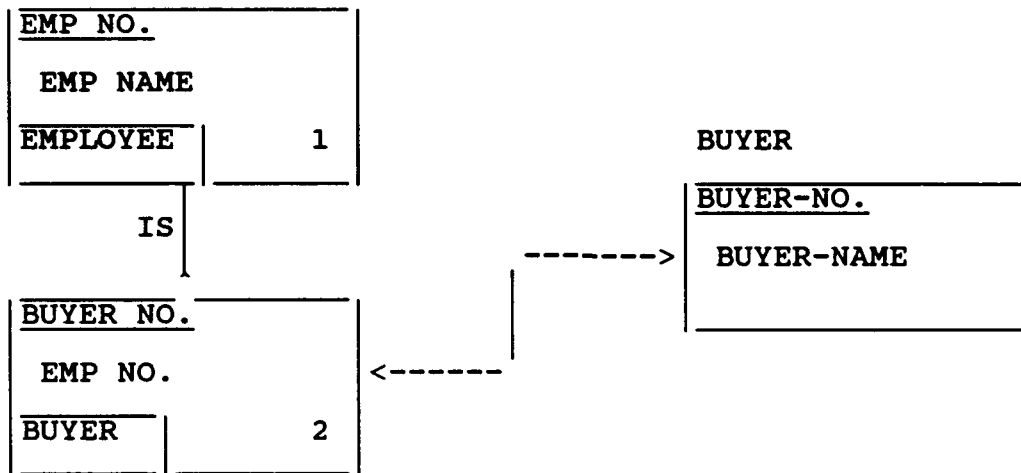
If the relation classes cardinality is one-to-zero-or-one, the concentration can be done in either direction, independent to dependent or dependent to independent, because neither can cause a repeating data field. The second and third examples in Figure 7-4 show these two situations. In the second, there is one BUYER user view row for each Buyer entity instance, and there is no row for an employee who is not a buyer. In the third example, there is one EMP-MAST instance for each Employee

instance. If an employee is not married, the SPOUSE-NAME data item in the user view row for that employee contains a null value.

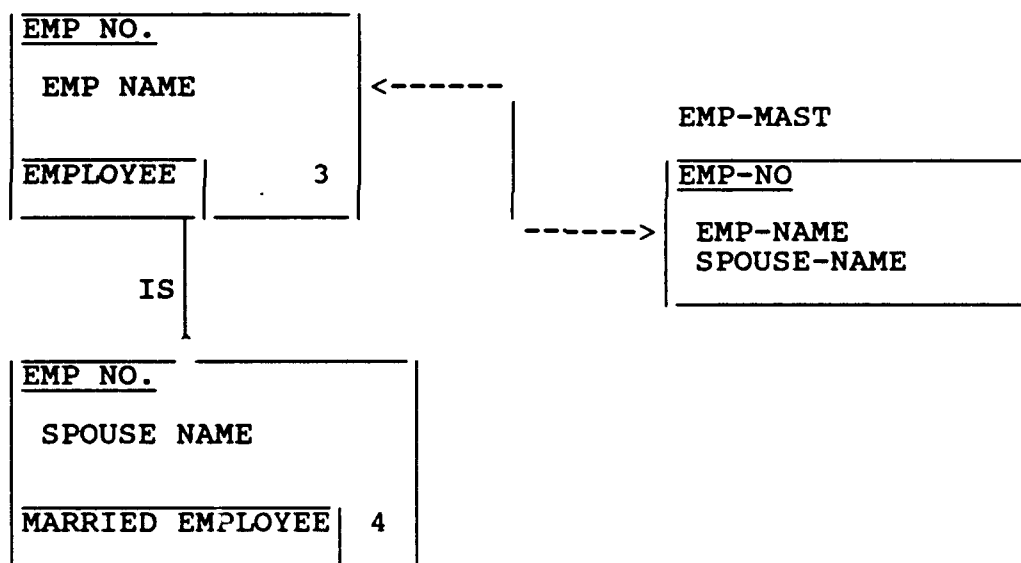


Example 1: ONE-TO-MANY RELATION CLASSES

Figure 7-4. ES-CS Join Examples



Example 2: ONE-TO-ZERO-OR-ONE RELATION CLASS



Example 3: ONE TO ZERO-OR-ONE RELATION CLASS

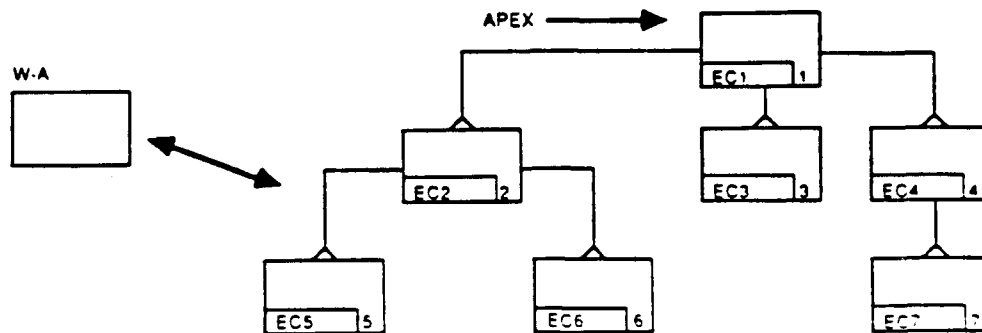
Figure 7-4. ES-CS Join Examples (Continued)

If the data items in a user view map to attribute use classes in several entity classes, they must all be combined to form the user view. This is done with a series of the join operations described above, each of which combines two of the entity classes. All of the entity classes must be interrelated such that they form one of the following (See Figure 7-5):

1. A regular hierarchy, that is, a structure in which:
 - o One entity class, called the apex, is not dependent on any of the others (e.g., EC1).
 - o Every other entity class is dependent on exactly one entity class (not necessarily the same one for all).
 - o Every relation class cardinality is one-to-zero-or-one.
2. A confluent hierarchy (an upside-down hierarchy), that is, a structure in which:
 - o One entity class, called the apex, has none of the others dependent on it (e.g., EC14).
 - o Every other entity class has exactly one entity class dependent on it (not necessarily the same one for all).
 - o . Any specific relation class cardinality is permitted.
3. A combination of
 - o One confluent hierarchy
 - o One or more regular hierarchies, each of whose apex entity classes is also in the confluent hierarchy (e.g., EC15, EC20, and EC25).

Each hierarchy is called a join structure. As shown in the examples in Figure 7-5, the user view must have one row for each instance of the apex entity class of the regular or confluent hierarchy. If a combination of hierarchies exists, the user view must have this correspondence to the apex of the confluent hierarchy.

REGULAR HIERARCHY:



CONFLUENT HIEARCHY:

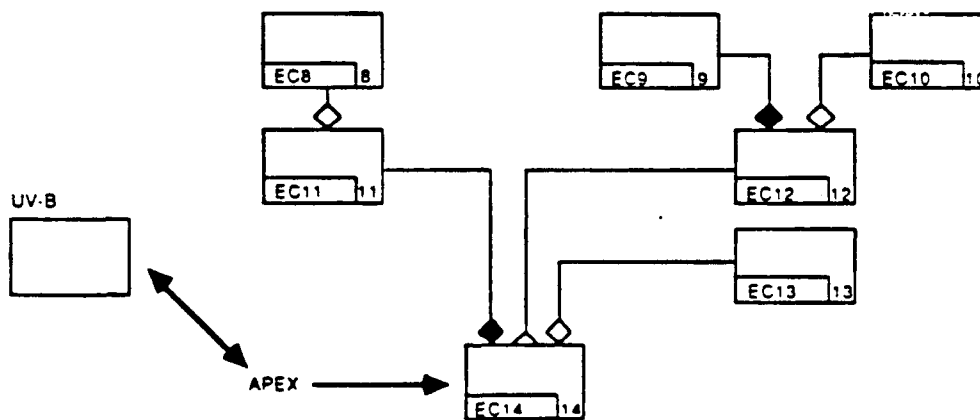


Figure 7-5. ES-CS Join Structures

COMBINATION:

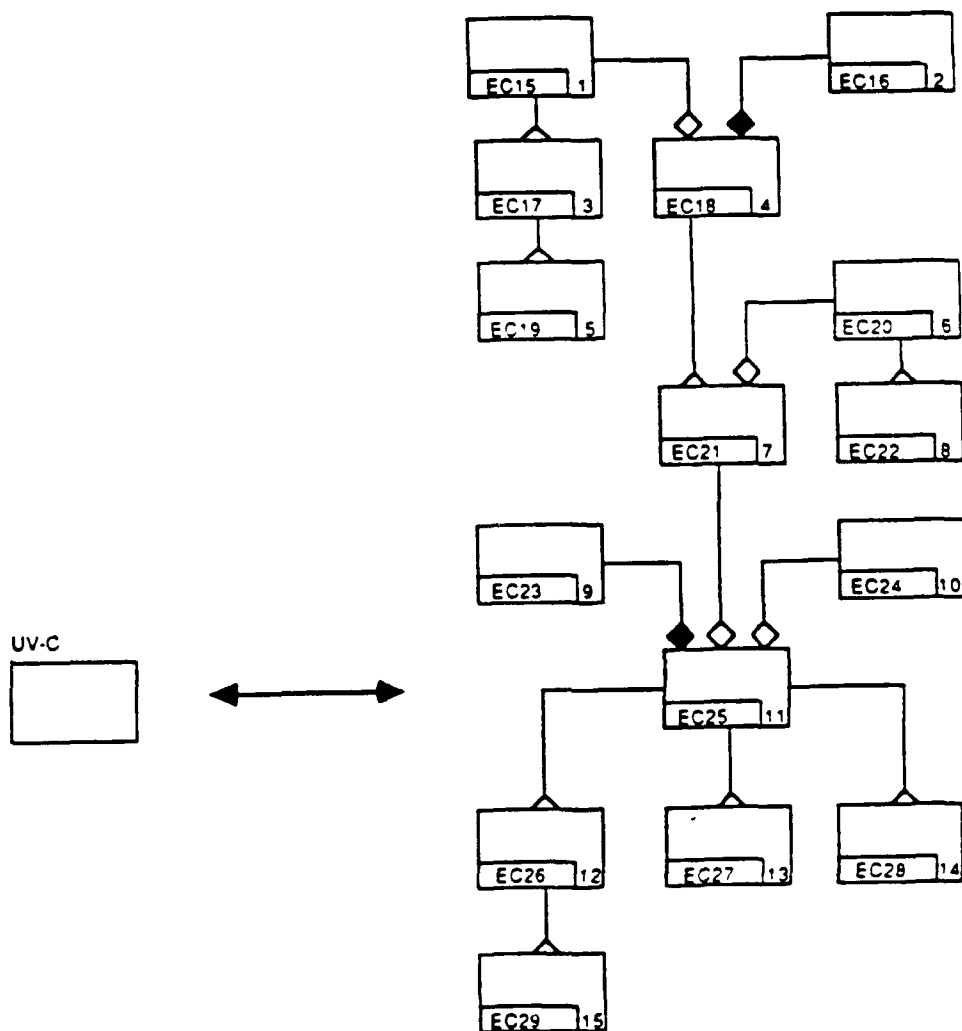


Figure 7-5. ES-CS Join Structures (Continued)

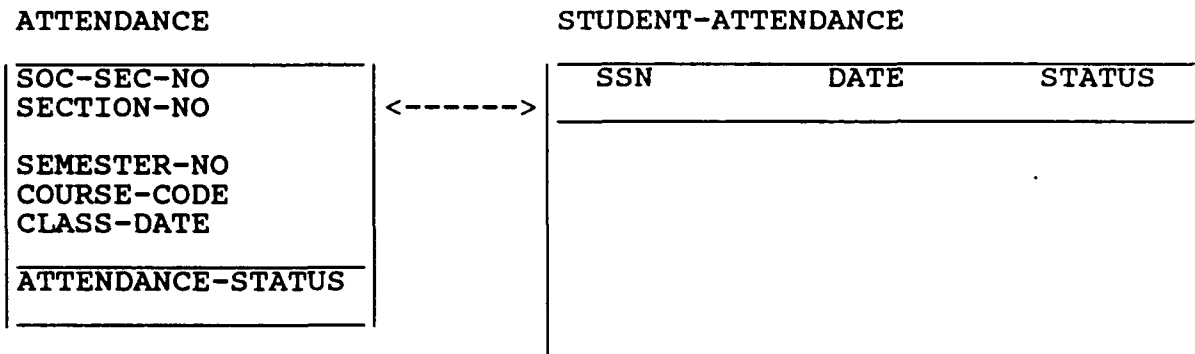


Figure 7-6. Single Entity Views

7.1.2 Modeling Forms

The following forms were developed to assist the CDM Administrator in determining the mappings between external and conceptual schema objects. If the view to be defined to the CDM is a single entity view, no modeling forms are needed to assist in the CS-ES mapping. Only one attribute use class exists which represents the data item of the user view. In Figure 7-6, the attribute use class SOC_SEC_NO is the only attribute in the entity ATTENDANCE that represents the data item SSN in the view STUDENT-ATTENDANCE.

When more than one entity is involved in the use view composition (see Subsection 7.1.1.2, Joins), the CDM Administrator needs to make choices as to which entity class the attribute use class is in, that corresponds to the data item instance. The forms listed below are used to assist the CDM Administrator to model mappings between these conceptual schema entities and the external schema view. They are:

User View Join Structure Diagram
Data Item/Attribute Use Class Mapping Form

In order for the CDM Administrator to define the mapping for each data item, he must first analyze what sort of data about the real-world things that the data item represents. Then determine which attribute use class in the conceptual schema represents the same sort of data as the data item. This involves finding the attribute use class whose definition or migration path corresponds to the intent of the data item. The first place to look is in the entity class that has one entity instance or each row in the user view. The value in the data item in each row of the view must be the same as the value in the attribute in the corresponding instance of the entity class.

If none of the attribute use classes in that entity class correspond to the data item, the next places to look are the entity classes that are related to that entity class. Again, the value in each view row must be the same as the value in the corresponding entity instance. If the attribute use class is

not in any of these entity classes, the search must be widened to include the entity classes that are related to them. This continues until the proper attribute use class is found or until it is determined that a new attribute class must be added to the conceptual schema (See Section 4.3). Once the proper attribute is found, fill out a line on the Data Item/Attribute Use Class Mapping Form for the attribute use class to which the data item maps.

The CDM Administrator determines if any joins are needed for the user view by investigating the possibility that data items in the user view map to attribute use classes in more than one entity class. This can be done by comparing the entity class names that are entered on the Data Item/Attribute Use Class Mapping Forms for the user view. If all the names are the same, the data items all map to attribute use classes in one entity class.

If the data items map to attribute use classes in more than one entity class, prepare a User View Join Structure Diagram. The entity classes must form one or more join structures as described in Section 7.1.1.2. If the join structures are not contiguous, one or more additional entity classes will be needed.

Instructions on how to fill out the user view forms follow.

User View Join Structure Diagram

Purpose: To provide a single source of information about the join structures for a user view.

Instructions:

Diagram the join structure for a user view which consists of two or more entity classes.

| <u>Form Area</u> | <u>Explanation</u> |
|-------------------|---|
| 1. User View Name | Unique identification name assigned to the use view by the CDMA. |
| 2. (Diagram Area) | Depiction of the entity classes and relation classes that make up the join structure. |

Data Item/Attribute Use Class Mapping Form

Purpose: To provide a single source of information about the mappings between external schema data items and conceptual schema attribute use classes.

Instructions:

Fill in for each user view the attribute use class that each data item maps to.

| <u>Form Area</u> | <u>Explanation</u> |
|------------------------|--|
| 1. User View Name | Unique identification name assigned to the use view by the CDMA. |
| 2. Data Item Name | Name by which the user identifies the data item. |
| 3. Data Type Name | The data format of the data item. Specify one of the six data formats supported by NDDL. |
| 4. Entity Class Name | Name of the entity class that contains the attribute use class being mapped to. |
| 5. Attribute Use Class | Name of the attribute use class to which the data item maps. |

7.2 Loading the Initial ES & CS-ES Mapping Definition

The external schema's objects are defined to the CDM in the following order:

User Defined data types (if different from the standard data type)
User Views
Data Items

The NDDL commands that define these objects and CS-ES mappings to the CDM are:

- a) ALTER DOMAIN
- b) CREATE VIEW
- c) DEFINE MODULE
- d) DEFINE ALGORITHM

7.2.1 Loading User-Defined data types

A domain can have several different styles for representing its values. These styles are defined with data types. A domain always has one standard data type that represents conceptual schema attributes. The format of the standard data type is limited to character, signed and unsigned formats. Many times it is necessary to describe the data storage representation of data items differently than the standard data type representation, either for presentation purposes or to support the NDML host language. Other data formats are float, integer, and packed. If a data item differs in format from the standard data type's definition, another user-defined data type can be added to the domain with the NDDL ALTER DOMAIN command. Figure 7-7 illustrates the CDM tables that are populated by the NDDL ALTER DOMAIN command and the relationship between USER_DEF_DATA_TYPE and DATA_ITEM.

The ADD TYPE clause of the ALTER DOMAIN command adds an entry to the USER_DEF_DATA_TYPE CDM Table with the same DOMAIN_NO the CDM assigned when the Domain was originally created. The user-defined data type's name, type, size, and

number of decimals are populated and the DATA_TYPE_IND is set to "user". The relationship between this data type and the data item is established when the actual user view is created.

7.2.2 Loading User Views and Data Items

A user view is a group of data items that a user wants to deal with as a group. The data items are associated with entity's attributes in the conceptual schema. The most simple form of a user view is when the attributes that map to data items of the view are all contained in one entity. The CDM Administrator loads this view into the CDM Tables by using the NDDL CREATE VIEW command. Figure 7-8 contains the CDM Tables populated by the CREATE VIEW command. An entry is added to the USER VIEW CDM Table with an assigned VIEW_NO and the VIEW_ID is populated with the given identifying name. The CDM administrator referred to the Data Item/Attribute Use Class Mapping Form for the Data Type Name. If the data items in the view are to be given names and data type formats other than that of the attribute use class, then the DATA_ITEM and data type clauses are used. The DATA_ITEM CDM Table is populated with the data item and data type name stated in this clause. If the clause is omitted, the attribute use classes and standard data type names associated with the attributes specified in the AS SELECT clause are used to populate the DATA_ITEM Table.

The AS SELECT clause of the CREATE VIEW command allows specific attributes to be selected from the entity. If all attribute use classes of an entity are desired, the CDM Administrator uses the AS SELECT * clause. This clause along with the FROM clause populates the PROJECT_DATA_ITEM CDM Table which actually defines the CS-ES mapping to the CDM. An entity is added to the VIEW_EC_XREF CDM Table for every entity specified on the FROM clause.

If the user view being loaded is a join, other optional clauses of the CREATE VIEW command are used. The CDM Administrator refers to the Data Item/Attribute Use Class Mapping Form for the data item names and the attribute use class names they map to. The data item names are supplied to the DATA_ITEM clause and the attribute use class names of the attributes the data items map to are listed in the AS SELECT clause of the CREATE VIEW command. The multiple entities that comprise the join are listed on the FROM clause. In order to join the entities, the WHERE clause is used so that their entity instances can be matched by equating the key class of the independent entity and the corresponding inherited attribute use class of the dependent entity. The User View Join Structure Diagram is referenced when equating the key classes. Besides allowing equi-join and/or outer-join conditions to be specified between one, some or all key class member of related entities, the WHERE clause can specify qualification criteria. The qualification criteria restricts the rows of entity instances that a user is allowed to access when utilizing the view in a program. An entry is created in the VIEW_QUALIFY_CRITERIA CDM Table for each item stated in the WHERE clause. If an attribute is named in the WHERE clause, an entry is added in the VIEW_QUAL_XREF CDM Table for each attribute.

The CDM Administrator can load descriptions for the user views with the NDDL DESCRIBE command with an object identifier of "VIEW". Descriptions for data items can be loaded with the same NDDL command and an object identifier of "DATA ITEM".

7.2.3 Loading Transformation Algorithms

When a complex mapping exists between a dataitem and an attribute use class, a software module can be invoked to provide the mapping. Examples of a complex mapping are: date transformations, unit of measure conversions, calculated fields, etc.

The software module takes as its input a dataitem, any number of constants and outputs an attribute use class. A reverse algorithm can also be written to take as its input the attribute use class, any number of constants and output a dataitem.

The NDDL DEFINE MODULE defines a software module listing the input and output parameters along with the data types. The CDM Tables populated are SOFTWARE_MODULE and MODULE_PARAMETER.

The NDDL DEFINE ALGORITHM command defines the use of a software module as a complex mapping algorithm between a dataitem and attribute use class. The CDM Tables populated are AUC_PARM and DI_PARM.

Figure 7-7 contains the CDM tables populated by the DEFINE MODULE and DEFINE ALGORITHM commands.

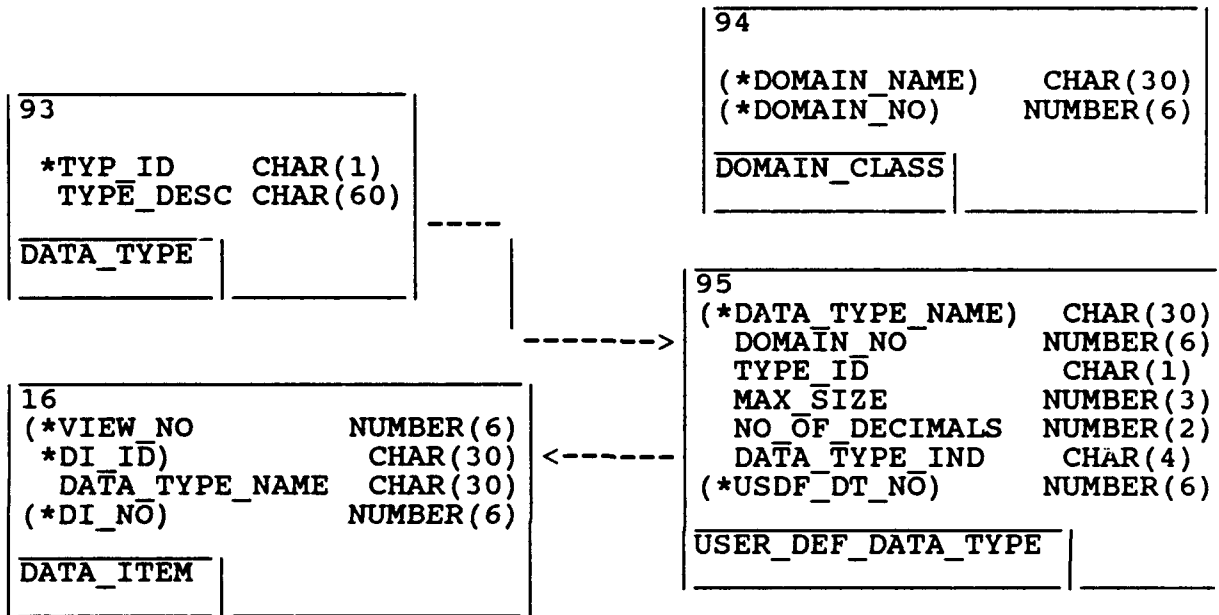


Figure 7-7. Domains and Data Types External Schema

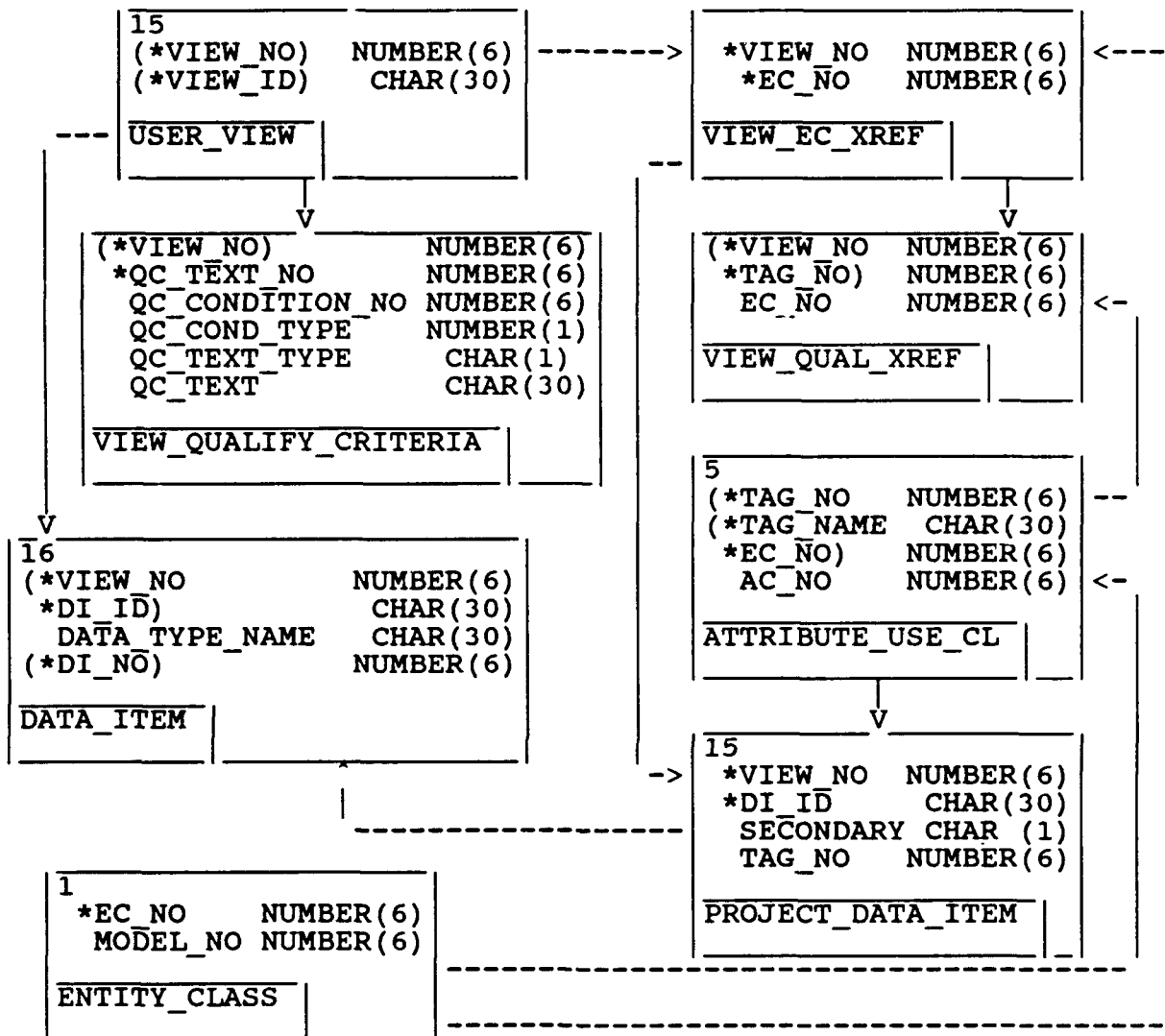


Figure 7-8. External Schema and CS/ES Mapping

7.3 Modifying/Deleting ES Elements and CS-ES Mappings

Prior to modifying or deleting objects of the external schema (i.e., user views) or the CS-ES mappings, the CDM Administrator must assess the impact of the proposed change on the other components of the CDM. The CDM Impact Analysis Utility identifies and reports which software modules are affected by a change to the CDM and also identifies and reports affected user views used by these software modules. Whenever changes are to be made to the external schema, a CDM Impact Analysis should be run to generate reports giving necessary information as to what additional action must be taken. The objective of this section is to provide the CDM Administrator with the information necessary to change a user view and determine the prerequisites before changing ES objects.

The NDDL commands to change and delete the external schema objects and its mappings are:

```
ALTER DOMAIN
DROP VIEW
DROP ALGORITHM
DROP MODULE
```

From this command list, it is evident that no ALTER VIEW NDDL command exists. In order to change a user view, the algorithms, if any are dropped, the view is then deleted and then re-created.

7.3.1 Modifying User-Defined data types

The user-defined data type allows the CDM Administrator to define to the CDM the data storage representation of a data item. This alleviates the restriction of using only the formats allowed for the standard data type. If the format representation of the data item changes, the data type originally specified for the data item can be changed to another data type. This is accomplished by dropping the user view and re-creating it. Another option available to the CDM Administrator is to change the actual format for the original data type that was specified for the data item. This is accomplished with the ALTER DOMAIN NDDL command. The ALTER TYPE clause allows the data type to be changed to another legal type with a new size and decimal specification. The USER_DEF_DATA_TYPE CDM Table is modified with the new information provided on the ALTER TYPE clause.

Any application programs that use this user view and generate application programs must be re-precompiled. These software modules will be specified on the CDM Impact Analysis reports.

7.3.2 User View Changes/Deletes

As was mentioned before, the CREATE VIEW NDDL command not only created the user view and data items but mapped the data items to attribute use classes. Likewise the DROP VIEW NDDL command deletes the user view and its CS-ES mappings. This is

the only way to change a user view's characteristics, by dropping the view and re-creating it. Before the user view can be deleted:

- * Drop the complex mapping algorithm that use data items of the user view being dropped, if any. Use the NDDL DROP ALGORITHM command.
- * Drop all software modules that use the external views reported on the CDM Impact Analysis Report.

Use the NDDL DROP MODULE command.

When a user view is deleted with the NDDL DROP VIEW command, all entries for the user view and its data items will be dropped in the USER_VIEW, DATA_ITEM, PROJECT_DATA_ITEM, VIEW_QUALIFY_CRITERIA, VIEW_EC_XREF, and VIEW_QUAL_XREF CDM Tables. All descriptive text for the view and its data items will be deleted from the CDM.

To re-create the view, refer to Subsection 7.2.2, "Loading User Views and Data Items."

APPENDIX A

GLOSSARY

Alpha-Numeric Data Format

A data format for values that can contain characters other than numerals (0-9). Numerals may be permitted also.

Attribute Class

A collection of all the same kind of attributes, i.e., those that have the same meaning. An attribute is a characteristic or fact about an entity. An attribute consists of a name (e.g., employee hire date) and a value (e.g., 15 August 1980). An attribute value may be:

- A. Nondivisible (e.g., state name)
- B. Divisible, i.e., a concatenation of two or more other attribute values (e.g., part number formed by concatenating drawing number and material code).
- C. Computed from one or more other attribute values (e.g., age computed as current date minus birth date).

Attribute Class Data Description

A generic data description that applied to a particular attribute class.

Attribute Use Class

A model attribute class that appears in a model entity class. Each attribute use class represents either an owned attribute class or an inherited attribute class.

Attribute Use Class/Data Field Mapping

Indicates that an attribute use class corresponds to a data item, i.e., that they have the same meaning and that the data item can be used to access the values for the attribute use class.

Attribute Use Class/Data Item Mapping

Indicates that an attribute use class corresponds to a data item; i.e., that they have the same meaning and that the data item can be used to access values for the attribute use class.

Attribute Use Class/Internal Schema Mapping

Indicates that an attribute use class corresponds to some portion of an internal schema.

Attribute Use Class/Record Set Mapping

Certain attribute use classes can be represented in a database by a group of record sets rather than be a data field. For example, Project: Task record sets called Pending, In-Process, On-Hold, and Completed. An attribute use class/record set mapping indicates that a particular record set corresponds to a particular attribute use class value.

Component Data Field

A data field that is part of another data field; e.g., if data field A is made up of data fields B, C, and D, each of these latter data fields is a component of A. A data field cannot be a component of more than one other data field.

Component Domain

An elementary domain that is part of another domain; e.g., a Date domain might be made up of a Month domain, a Day of Month domain, and a Year domain. Each of these latter domains would be a component of the Date domain. An elementary domain can be a component of several other domains.

Component Unit of Measure

An elementary unit of measure that is part of another unit of measure; e.g., the "inch" unit of measure is a component of the "foot-inch" unit of measure. An elementary unit of measure can be a component of several other units of measure.

Conceptual Schema

The description of all the shared data items within an enterprise's databases and of the allowable operations on and integrity constraints for those shared data items. Represented by a fully normalized information model in which integrity constraints have been completely specified. Not influenced by any usage or storage considerations. A software module that must be used to access or transform data that is stored in a manner that the CDMP is not designed to handle.

Constraint Statement

One complete NDDL description of either an assertion, a trigger, or a horizontal partition fragment. An assertion is a rule about values for attribute use classes. If an NDML command attempts to violate an assertion, the CDMP rejects the command with an error message. A trigger is a set of conditions and a set of actions, both involving entity classes and attribute use classes. If the conditions are satisfied all the actions are taken. If the conditions are not satisfied, none of the actions are taken. See the definitions of Horizontal Partition and Horizontal Partition Fragment for details about this use of constraint statements.

Database Area

A subdivision of a CODASYL database. This subdivision is a technique for improving the efficiency accessing database record type instances. When a database is subdivided into database areas, some or all of its records types are assigned to particular areas. Instances of these record types are stored only within the assigned areas. Then, these record type instances can be accessed by searching only the appropriate areas rather than the entire database. This access method is only used when the record type instances cannot be located by other means (e.g., by calc keys or record sets).

Database Area Assignment

Indicates that a record type is assigned to a database area.

Database Directory

A software library that must be used when accessing a database.

Database Password

A code that must be supplied when logging on to a DBMS to use a database. The DBMS verifies the password before accepting any other messages.

Data Field

A portion of a record type in which data values can be stored.

Data Field/Record Set Linkage

A data field in a variable data set in a TOTAL database that is used as the variable control key for a linkpath from a master data set.

Data Field Redefinition

A data field that occupies the same space in a record type as another data field. A record instance cannot contain values in both data fields. One instance can contain a value in one field while another contains a value in the other.

Data Format

The portion of a generic data description that includes the structural characteristics such as data type, length, storage method, etc. If a generic data description is for elementary values (e.g., customer names), it will have only one data format (e.g., Data Type - alphanumeric, Length = 30). If it is for compound values (e.g., part numbers consisting of six numerals followed by three letters followed by four more numerals), it

will have more than one data format, one for each elementary portion of the values. For the part number example the data formats would be:

- | | | |
|----|------------------------|------------|
| 1. | Data Type = numeric | Length = 6 |
| 2. | Data Type = alphabetic | Length = 3 |
| 3. | Data Type = numeric | Length = 4 |

A generic data description with a compound unit of measure, i.e., one that is a group of component unit of measures, must have a data format for each component unit of measure.

Data Item

An attribute class as seen by a user in a user view, i.e., a kind of data (e.g., employee hire date), not a particular data value (e.g., 15 August 1980).

Data Management System

Either a database management system or a file management system, i.e., a set of computer programs that must be used to establish and maintain a database or a computer file.

Data Type

The combination of a type of values (e.g., alphanumeric, signed numeric, etc.) and a type of storage (e.g., binary, packed, etc.)

Dependent Entity

The entity class that is dependent in a specific relation class. A dependent entity, i.e., an entity is a dependent entity class, can exist only if it is related to an independent entity. Contrast with independent.

Description Type

A generic object may have several different kinds or styles of description (short, long, technical, nontechnical, etc.). Each is a description type.

DMS on Host

A data management system that is available on a particular host.

Domain

A set of rules about the values that are allowed for a data item, attribute class, or data field. A domain is either an elementary domain or a group of two or more elementary domains, called component domains.

Domain Range

A series of consecutive values that represent all or part of an elementary domain.

Domain Value

A single value within an elementary domain.

Elementary Data Field

A data field that does not have any component data fields.

Elementary Domain

A domain that does not have any component domains. An elementary domain can be expressed as a series of values or value ranges.

Elementary Unit of Measure

A unit of measure that does not have any component units of measure.

Entity Class

A collection of similar entities, i.e., those that have the same kinds of attributes. An entity is a person, place, event, thing, concept, etc.

Entity Class/Record Type Join

A relational join operation that combines two related entity classes as part of the design of a record type.

Entity Class/Record Type Mapping

Indicates that an entity class corresponds to a record type, i.e., that they both have the same meaning and that the record type can be used to store instances of the entity class.

If a record type has more than one EC-RT mapping, some of its instances correspond to instances of one entity class while others correspond to instances of another, i.e., the record type is the relational union of the entity classes. An example is a Replenishment Order record type that maps to both the Purchase Order and Manufacturing Order entity classes. Each record instance represents either a purchase order or a manufacturing order.

Entity Class/Record Type Union Discriminator

If a record type corresponds to more than one entity class, i.e., if it has more than one EC-RT mapping, it is the relational union of those entity classes. Some instances of such a record type correspond to instances of one of the entity classes, others to those of another. For such a record type there must be a way to determine which record instances correspond to instances of each entity class. An entity class/record type union discriminator provides this by specifying that a given value in a given data field indicates that a given EC-RT mapping should be used.

Entity Class/User View Join

A relational join operation that combines two related entity classes as part of the design of a user view.

External Schema

See User View.

File

A set of stored data that is managed by a file management system (e.g., VSAM).

File/Database

A set of stored data, i.e., either a computer file (e.g., a VSAM or flat file) or a database (e.g., an ORACLE or IMS database).

Generated Request Processor

A software module that was created by the CDMF Precompiler.

Generic Data Description

A detailed description of the values for one or more data items, attribute classes, data fields, and/or module parameter. It includes format, measurement, and domain characteristics of the values.

Generic Data Description Component Unit of Measure

A component unit of measure that is specified as part of a data format. These are only specified for a generic data description that includes a compound unit of measure, i.e., one that is a group of component units of measure.

Generic Data Description Domain

A domain that is specified as part of a generic data description.

Generic Data Description Unit of Measure

A unit of measure that is specified as part of a generic data description.

Generic Object

Anything with a name that distinguishes it from other things of the same type and with a description that explains what it is; e.g., any entity class or attribute class.

Generic Object Description

An explanation of what a particular object is.

Generic Object Description Line

One fixed-length portion of a generic object description.

Generic Object Keyword

A keyword for a particular generic object.

Generic Object Name

An noun or noun phrase by which a generic object is known. Two objects can have the same name.

Horizontal Partition

Indicates that the same record type is not used to store all instances of an entity class, i.e., that one is used to store some instances while another is used to store others. Each record type represents a "fragment" of the entity class. These fragments do not overlap, i.e., no entity instance appears in more than one fragment. An entity class can be partitioned into any number of fragments, usually with each being in a different [Bdatabase or file, although that is not a requirement; some or all may be stored as different record types in the same database or file. A constraint statement defines each fragment, i.e., describes the conditions that must be met by each entity instance that is stored as a given record type. If an entity class is replicated, i.e., if each of its instances is stored in more than one database instances is stored in more than one database or file, each replication can be horizontally partitioned. For example, for the first replication the instances could be partitioned based on the values in one attribute use class, and for the second replication they could be partitioned based on the values in another.

Horizontal Partition Fragment

A record type that is used to store some, but not all, of the instances of an entity class. A constraint statement describes the conditions that must be met by each entity instance that is stored as the record type. If the conditions are satisfied by the attribute values of an entity instance, it can be stored as an instance of the record type; otherwise, it cannot be.

Host

A computer in the IISS.

IMS Segment

A record type in a database that is controlled by IBM's IMS DBMS.

Independent Entity

The entity class that is not dependent in a specific relation class. An independent entity, i.e., an entity in an independent entity class, can exist without being related to a dependent entity. Contrast with dependent entity class.

Inherited Attribute Class

An attribute class that appears in a dependent entity class because it has migrated from an independent entity class. Must be part of a key class in the independent entity class.

Inherited Attribute Classes Form

Provides a single source of information about inherited attribute use classes that are to be described in the conceptual schema.

Inherited Key Class

A key class in the independent entity class of a relation class that has migrated to appear in the dependent entity class of that relation class.

Internal Schema

A description of the data items in a database. Described from DBMS User's perspective. Usually not fully normalized.

Join

A relational operator that creates a new relation by combining two or more source relations according to specified criteria. A natural join combines the relations by matching tuples with equal values for a common attribute class (column).

Key

An assortment of attributes in an entity that can be used to uniquely identify that entity within its entity class. An entity can have more than one key; e.g., an employee can be uniquely identified by either an employee number or a Social Security Number.

Key Class

A group of one or more of an entity's attributes that can be used to uniquely identify the entity within its entity class. An entity can have more than one key. A key class is a collection of the attribute classes whose member attributes comprise the keys for the entities in an entity class. An entity class has the same number of key classes as each of its member entities has keys. For example, if each entity has three keys, the entity class has three key classes.

Key Class Member

An attribute use class that is part of a key class.

Key Class Migration

The process of moving key classes from independent to dependent entity classes.

Library Module

A software module that is stored in a software library.

Model

A representation of the information requirements of all or part of an enterprise in terms of entity classes, relation classes, and attribute classes.

Model Glossary Name

A name of a model entity class or a model attribute class, either an official name or an alias.

Module Parameter

A means of supplying values to a software module and of receiving results from a module.

Numeric Data Format

A data format for values that can only contain numerals (0-9) and associated punctuation (decimal point, comma, etc.).

Owned Attribute Class

An attribute class that is not an inherited attribute class.

Owned Attribute Classes Form

Provides a single source of information about owned attribute use classes that are to be described in the conceptual schema.

Program Control Block

A portion of a PSB that describes and controls how an IMS database can be accessed.

Program Specification Block

A group of logical views of IMS databases that is used for interacting with the IMS DBMS.

Record Set

An association between one record type, called the owner, and one or more other record types, called the members.

Record Set Member

A record type that is a member of a record set.

Record Type

A group of data values that are stored together as a unit in a computer file or database. A record type is the collection of all the records of the same kind, i.e., all the records that contain the same kind of data values.

Relation Class

An association between an entity in one entity class and one in another. A relationship has a label that describes the association. For example, a customer named ABC Corp. is associated with an order numbered 123 in a manner labeled "placed". A relation class is a collection of the identically labeled relationships between the members of the same two entity classes. Each relation class is either specific or nonspecific.

In a specific relation class, one entity class is "independent" while the other is "dependent"; i.e., entities in the first can exist without being associated with any in the second, but those in the second cannot exist without being associated with one in the first. One key class from the independent entity class "migrates" through each specific relation class to appear in the dependent entity class as inherited attribute classes.

In a nonspecific relation class, neither entity class is dependent on the other; i.e., entities in either entity class can exist without being associated with any in the other. For convenience, one entity class is arbitrarily called "independent" and the other is called "dependent".

Relation Class Form

Provides a single source of information about relation classes that are to be described in the conceptual schema.

Relation Class/Record Set Mapping

Indicates that a record set represents the same association as a relation class. If a record set has more than one member record type, it may represent several relation classes, a different one for each member. Hence, this entity class is only indirectly dependent on record set (via record set member).

Repeating Data Field Occurrence Counter

A data field whose data values indicate how many occurrences of a repeating data field actually contain values.

Segment Data Element

A data field is an IMS segment.

Software Library

A computer file in which software modules can be stored.

Software Module

A set of computer instructions that are treated as a whole, i.e., stored, compiled, and executed together.

Subschema

The description, in the DDL of a CODASYL DBMS, of all or part of a database. For IISS, only one subschema is needed for a CODASYL database, and it must describe all the common data within the database that is to be accessible with NDML.

Unit of Measure

A standard scale for determining the magnitude of something. Examples include inch, foot, foot-inch, meter, ounce, pound, hour, minute, second, etc.

Unit of Measure Conversion

A means of transforming a value expressed in one unit of measure into an equivalent value expressed in another; e.g., transforming inches to feet or feet to meters.

Unit of Measure Conversion Constant

A number in a unit of measure conversion step that is the same every time the conversion is performed. A software module that can be used to perform a unit of measure conversion. A module parameter that is used to supply values to or receive values from a unit of measure conversion module.

Unit of Measure Conversion Step

One of a series of arithmetic steps that can be used to perform a unit of measure conversion. Each step takes the value resulting from the prior step (the first step uses the value to be converted) and adds, subtracts, multiplies, or divides by another value, either a constant or a variable. The result of the last step is the converted value. The processing sequence is always first steps to last; parentheses, branching, and conditional tests are not allowed. Consequently, some unit of measure conversions cannot be performed in this manner; e.g., converting meters to feet-and-inches.

Unit of Measure Conversion Variable

A number in a unit of measure conversion step that can be different every time the conversion is performed. This is only used when the unit of measure being converted from has two or more component units of measure. Each component is a variable and each is involved in a separate step.

User Application Process

A software module that supports business activities rather than data processing activities and that can be executed directly, i.e., a main routine, not a subroutine. A user AP may contain NDML commands for accessing stored data via the CDM, or it may access them directly via DMSs, or it may call subroutines that contain NDML commands or that access stored data directly.

User View

A group of data items that a user wants to deal with as a group. It is similar to an entity class but does not necessarily meet all the conditions for being one, it can be thought of as an unnormalized entity class. A user view is often the result of combining several entity classes via relational join operations and selecting particular attribute use classes as data items via relational project operations.

Vertical Partitions

An entity class is vertically partitioned when some of its attribute use classes map to data items in one user view and others map to those in another. An entity class can have several vertical partitions.

APPENDIX B
REFERENCES

ICAM Life Cycle Documents

FTR11021000U Volume V, Information Modeling Manual
 (IDEF1)

PRM620341200 Embedded NDML Programmers Reference Manual

UM620341100 Neutral Data Definition Language (NDDL)
 User's Guide

UM620341002 Information Modeling Manual - IDEF1-
 Extended (IDEF1X)

TBM62034100 CDM1 - An IDEF1 Model of the Common Data
 Model

Other References

"The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management System" American National Standards Institute, AFIPS Press, Montrole New Jersey, 1977.

Atre, S., "Data Base, Structure Techniques for Design, Performance, and Management", John Wiley and Sons, Inc., New York, 1980.

Martin, James, "Managing the Data Base Environment", Volumes I and II, Savant Institute, 1981.